Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingenería de la Computación

reCipe: A program synthesizer based on model checking for temporal logics

T E S I S

QUE PARA OPTAR POR EL GRADO DE:
Maestro en Ciencia e Ingeniería en Computación

PRESENTA:
Fernando Abigail Galicia Mendoza

TUTOR
Dr. David Arturo Rosenblueth Laguette IIMAS, UNAM

TUTOR
Dr. Armando Solar-Lezama
CSAIL, MIT

Ciudad Universitaria, Cd. Mx. Febrero 2021

Acknowledgments

First, I want to thank my parents, Leticia Mendoza and Ramón Galicia, for their constant support of my goals. I can achieve more academic credentials, but I am going to be still learning form both of you.

This work would not be the same without the feedback, teaching and support of my advisers David Rosenblueth and Armando Solar-Lezama. Your knowledge of computation, mathematics, language and other topics (like recipes) created an excellent work environment for the development of this work.

The feedback given by the committee formed by David Flores, Favio Miranda and Francisco Quiroz, not only enriched this work, but also, was an interesting challenge to do in this step of my academic life.

The moments I shared with my friends: Antonio, Estefanía, Joshua, and Montserrat, were the basis to keep trying to end this work without losing my mind. Besides the actual situation, you gave me part of your time for talking, laughing, etc. I really appreciate these kind of actions.

At last but not least, I want to thank to CONACyT (*Consejo Nacional de Ciencia y Tecnología*), under the national scholarship *CVU* 928466, and MIT (*Massachusetts Instiute of Technology*), for the economical support for the develop of the current thesis work.

Abstract

One of the goals in computer science is the automation of tasks. An especially important task is the writing of programs. More formally we can describe this problem as: *Given a high-level specification φ, find a program that satisfies φ.* This should not be confused with the problem of compiling a program. The difference comes from the word *find*; a compiler takes a high level description of the behavior of the program, then the compiler applies some transformation rules to generate machine code. The process we are exposing is *synthesis*, and more than a transformation over a high-level specification, the system finds a possible program that satisfies a user specification.

In this thesis, we introduce *reCipe*, a new style of synthesis that takes advantage of the temporal expressiveness given by the temporal logics formulas. This style differs from others in that it is focused in taking advantage of the semantics of a temporal logics. reCipe is based on prior work by Morgenstern and Schneider [19].

We use logic systems to formalize aspects of the entities that we work on in this thesis. The process of formalizing has such consequences in computer science that it has its own name: formal verification. On the one hand, we are interested in logic systems that can describe the behavior of a program as *states*. On the other hand, one of the interests of computational logic is to "model check" a system with respect to a property. We describe this last task as: *Given a structure and a specification, decide if the structure satisfies the specification.* Therefore, we are interested in programs that can solve the problem of model checking for *temporal logics*.

The problem that reCipe solves is how to generate a program from a user specification, where the user specification is a temporal formula with first-order quantifiers. Our procedure creates a sequential program that should satisfy the user requirements. One of the benefits of reCipe compared with other techniques lies in the idea of creating a program by only using the information given in a formal specification, and not the control structure.

The core of reCipe lies in the semantics of the temporal operators: EX and AX. The first one denotes the existence of a program P. The second one is used to analyze all possible inputs over P. In other words, we choose a program P and execute P with each possible input. In this last step, P should satisfy all possible inputs.

Contents

List of Figures

List of Codes

List of Sketches

List of Responses

Chapter 1

The Synthesis Problem

"The approach of (program) synthesis by theorem proving tell us that such a system need not to contain much more than a theorem prover."

Pnueli, Amir & Rosner, Roni (1989) [21]

In a general context, *synthesis* is the action of combining separate ideas, techniques, elements, etc. in order to get a new product. Therefore, we can describe the synthesis of a program as the search of techniques able to generate a program from a collection of semantic and syntactic requirements [24].

There are different approaches to solve the program-synthesis problem. For example:

- Deductive synthesis: given a program, the synthesizer generates a more efficient implementation.
- Inductive synthesis: given a set of inputs/outputs (IO), the synthesizer generates a function that matches with IO.
- Functional synthesis: generate a program from a declarative specification. Traditionally, the specifications for these synthesizers are $\forall\exists$-formulas or similar.
- Automaton synthesis: given a temporal formula with a specific form, synthesize a transition system that satisfies the temporal formula.

In this thesis, we extend the synthesis defined by Morgenstern and Schneider in [19]. Such a work is based on two aspects:

1. Use as specification an incomplete program with assertions, a *sketch* [23].
2. Use the semantics of temporal logics to give richer specifications over the execution of a program.

There are other implementations that solve the synthesis problem. One example is the program extraction tool of the proof assistant `Coq`[1]. One of the steps of this extraction tool is the transfor-

[1]Official website:`http://coq.inria.fr/`

mation of a proof into a λ-term. This *transformation* is a solution of functional synthesis [20].

In the next sections, we introduce two current synthesis approaches that use the two previous aspects. At the end of this chapter, we mention how Morgenstern and Schneider *combine* the previously aspects. We formalize the work by Morgenstern and Schneider in Chapter 3.

1.1 A program begins with a sketch

In this section, we introduce a state-of-the-art program synthesizer called Sketch. Created by Solar-Lezama in [23], Sketch is an implementation of the syntax and semantics of a language, with the same name, allowing a programmer to give an incomplete program with assertions that should be satisfied. Solar-Lezama called this type of templates *sketch*.

The synthesizer has the following behavior: it first parses a sketch and creates a set of constraints that are given to an *SMT solver*[1]. Then, the SMT solver searches for the values that will satisfy the constraints and finally, the synthesizer fills in the "holes" of the incomplete program.

To deal with the synthesis problem, Sketch represents the problem as follows: Let us use φ to denote the description of the missing parts of the program (called *holes*), σ to denote the input state of the program and Q to denote a predicate that is true iff the complete program satisfies φ under the input σ. Therefore, Sketch solves the program-synthesis problem by solving the following constraint:

$$\exists \varphi . \forall \sigma . Q(\varphi, \sigma)$$

Solar-Lezama in [23] explains that the synthesizer does not try, in one attempt, to search for a φ that works for all possible inputs. Instead, Solar-Lezama defines the procedure *Counterexample Guided Inductive Synthesis* (CEGIS) in order to perform a smarter search. This approach has the following behavior (see Figure 1.1): the synthesizer generates a new candidate φ' for a particular input σ', then the synthesizer verifies if $Q(\varphi', \sigma'')$ holds, where σ'' is a new state of the input. If the verification fails, the synthesizer will add σ'' into the search space, and repeats the procedure. This procedure ends with either a program that satisfies all the inputs or with a message saying that the synthesizer was not able to generate a candidate with the allotted resources (i.e., time and memory).

In Figure 1.2, we show a toy example of a sketch and the completed program generated by Sketch. In this case, the synthesizer figures out that one possible value for the hole (`??`) is the number 1.

Morgenstern and Schneider try to extend the domain of the `assert` statement by adding temporal operators. To satisfy these new assertions, they suggest to replace the SMT solver by a model checker.

[1]A SMT solver is a program that proves the satisfiability of a SMT instance, where a SMT instance is a formula in first-order logic.

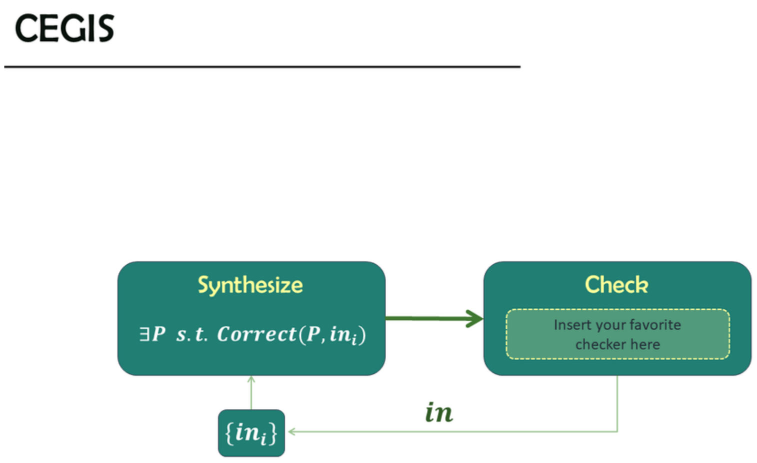
CEGIS
Synthesize
$\exists P\ s.t.\ Correct(P, in_i)$
Check
Insert your favorite checker here
in
$\{in_i\}$

Figure 1.1: A diagram of CEGIS created by Solar-Lezama in [24]

```
void decrement(int x){
  while(x > 0){
      x = x-??;
  }
  assert (x == 0);
}
```

Sketch 1.1: A sketch of the decrement function.

```
void decrement(int x){
  while(x > 0){
      x = x-1;
  }
  assert (x == 0);
}
```

Code 1.1: The completed sketch.

Figure 1.2: An example of a sketch and its completed version.

1.2 The power of the temporal specifications

Another approach that at first sight might appear to be slightly away from the program synthesis is the automata approach. This last approach exploits the semantics of the temporal specifications so as to generate a state machine that satisfies a temporal specification. We only present a brief idea of how this synthesis is done.

The pioneers of this approach are Pnueli and Roni in [21]. They have different synthesis goals in comparison with Solar-Lezama [23]. Pnueli and Roni try to synthesize reactive programs. The main difference between reactive and non-reactive programs is that the first ones are designed in order to run *all the time*. An example of this kind of programs is a program handling the access of users into a web page.

Pnueli and Roni take a temporal formula $\varphi(x, y)$, where x is an input and y is an output, and try to synthesize a state machine that satisfies $\varphi(x, y)$. In other words, they deal with a formula of the form:

$$\exists f.\forall x.\varphi(x, f(x))$$

where f is a function and x is an input variable.

Their approach is to split the variables into input and output variables. If there exists a system that realizes the specification, construct such system. The construction is based on a two-player game between the environment (input variables) and the system (output variables). The winning condition is when the *system* is capable of satisfying φ against all the possible input variables.

The researches working on reactive synthesis have presented different strategies to synthesize this type of systems. Examples of these strategies are: taking formulas that have a specific form or define new types of automatas so as to deal with the exponential time caused by the mentioned game.

However, the basic idea of this synthesis lies in the semantics of temporal logics. Morgenstern and Schneider add this expressiveness into the assertions used by Sketch.

1.3 The basis of two worlds

Morgenstern and Schneider in [19] extend the domain of the assert statements in order to use temporal operators. In Sketch 1.2, we show an example of this extension.

On the one hand, this extension allows us to give more powerful assertions against the original approach. On the other hand, the semantics of the temporal logics allows to specify concurrent behaviors. Therefore, in theory, we could synthesize sketches with concurrent behaviors. Morgenstern and Schneider argue that the only critical change between their approach against the approach of Solar-Lezama is the type of checker. Whereas Solar-Lezama uses a SMT solver, Morgenstern and Schneider use a model checker. They encode the process of completing a sketch into a *Kripke structure* and use the Kripke structure for verifying the assertions given by the user.

In Chapter 3, we present and extend the idea of Morgenstern and Schneider, in order to synthesize programs where the only input is a temporal formula.

1.3.1 reCipe: One step further

The procedure defined by Morgenstern and Schneider in [19] takes as input a sketch, where the assertions occurring in the sketch have temporal logic formulas, and returns a program that satisfies the assertions. In this thesis, we argue that such a procedure can be generalized by dispensing with the program structure determined by the sketch. More precisely, we contend that it is possible to generate an entire program by only using as the input a temporal logic formula.

```
void func(int x){
    while(x > 0){
        x = ??;
        assert (EF x == 0);
    }
}
```

Sketch 1.2: A sketch with a temporal assert.

Temporal logic formulas not only establish properties of each state of a program, but also describe the desired behavior. With the previous idea in mind, if we generate a sketch with holes that affect both the values of the variables and the program flow, then we will be able to use the procedure defined by Morgenstern and Schneider. A sketch of the program flow is a program with holes in the `goto`-statements, and holes in the guards of the `if`-conditionals.

In Figure 1.3, we give a diagram of our procedure: *reCipe*. First, we use the information from a user specification φ for generating a sketch S and a set of candidate expressions C that fill in S. After creating S and C, we create a structure M that encodes the process of filling in S, by using the expressions of C, and the execution of the generated programs. We verify if there is a encoded program behavior P in M that satisfies φ. If the verification succeeds, we can generate a program from P; otherwise, we create a new set of candidates C', that has more complicated expressions than C, and try again the process by using the same sketch S and replacing C by C'.

In Chapter 3, we give detailed information about the execution of Figure 1.3. In the next chapter, we give a brief introduction in how to solve the model-checking problem for a temporal logic whose semantics adjusts with our main goal.

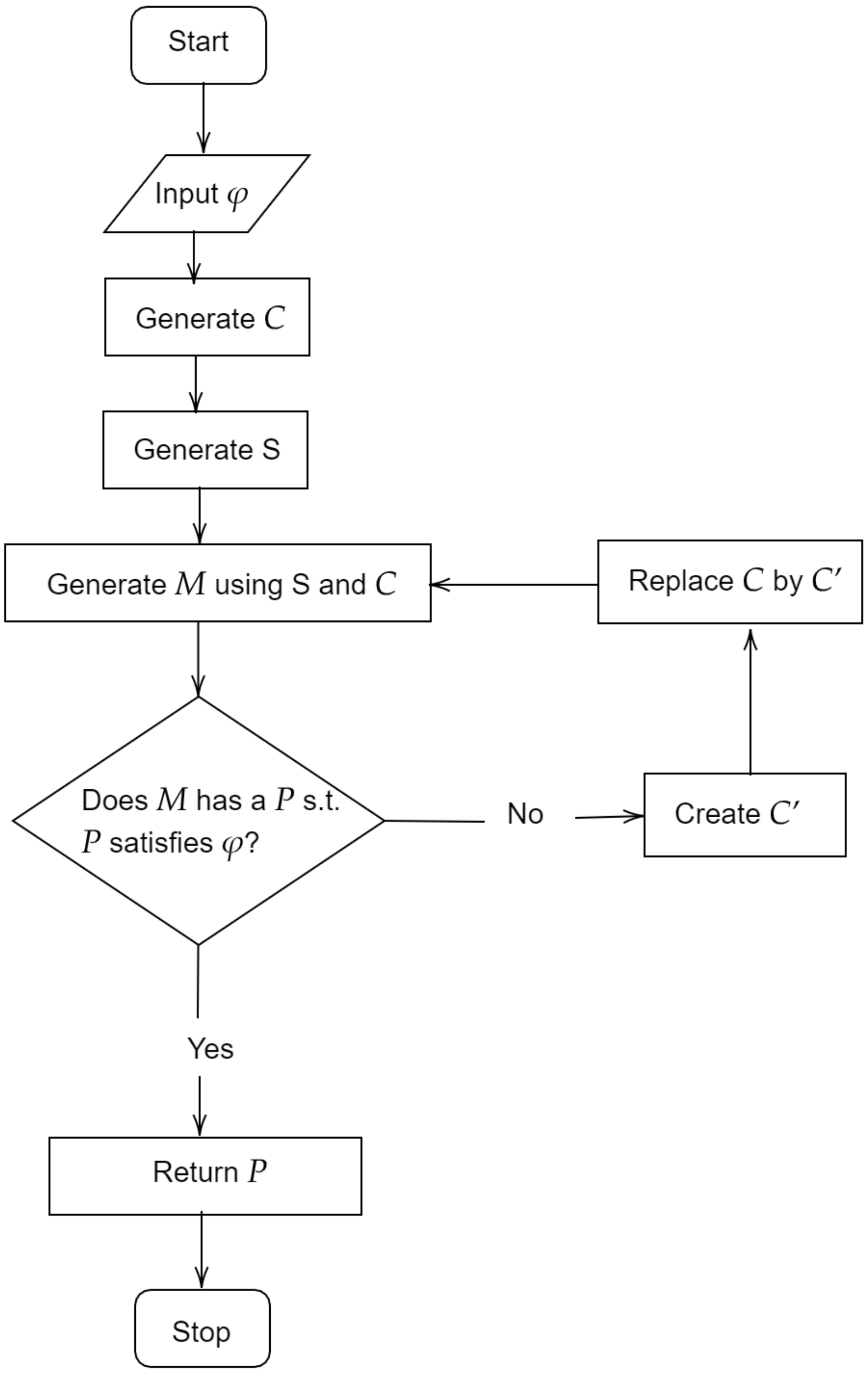

Figure 1.3: A diagram of *reCipe*.

Chapter 2

CTL model checking

"Companies such as AT&T,(. . .),Siemens, and Sun are using model checkers increasingly on their own designs to reduce time to market and ensure product quality."

Halpern, Joseph et al. (2001) [14]

In this chapter, we introduce the model-checking problem for *computation tree logic* (CTL) and one algorithm that solves the model-checking problem for CTL.

2.1 Introduction

As users, many of our personal and professional activities are assisted or done by automated systems. We take a leap of faith assuming the correctness and soundness of these systems [22]. Nowadays a large number of these results are "proved" by *testing*. However, another method has been used for software analysis: *formal verification*. In contrast with testing, in formal verification we explore all the possible scenarios and behaviors of a particular automated system through a formal logic system [9].

There exists a variety of methods in formal verification [13]. In the particular case of this thesis, we work with *model checking*; see Definition 2.1.

Definition 2.1 (Model Checking [1])**.** Let $\mathcal{M}$ be a model and φ be a formula. We define the model-checking problem as the process of verifying if the following holds:

$$\mathcal{M} \models \varphi$$

Definition 2.1 establishes two inputs in model checking: a model and a formula. The model and the property must be formalized through a logic system. In our case, we want a logic system with the following characteristics:

1. Computationally decidable.
2. Captures the temporal behavior of a software system.
3. Simulates *non-deterministic choices.*
4. There exists an implementation of the model checking procedure.

Computation tree logic (CTL) has the mentioned characteristics. In the next section, we introduce the syntax, the semantics and how to represent the model-checking problem for CTL.

2.2 Computation Tree Logic

A logic system provides us with the formal syntax for the abstraction of a property. In the case of *temporal logics*, we can formalize temporal behaviors of a system. *Computation tree logic* (CTL) is a temporal logic that allows us to model time as a tree-like structure [15]. Using the tree-like structure, we can model different possible futures as paths where each state in those paths is a node of a tree. Let us begin by introducing the syntax of CTL (see Definition 2.2).

Definition 2.2 (Syntax of CTL [15])**.** The syntax of CTL is constructed by the following Backus-Naur grammar:

$$\varphi \ ::= \ v \mid \neg\varphi \mid \varphi \wedge \varphi \mid AX\ \varphi \mid EX\ \varphi \mid EF\ \varphi \mid AF\ \varphi \mid E[\varphi\ U\ \varphi] \mid A[\varphi\ U\ \varphi] \tag{2.1}$$

where v belongs to a possibly infinite-countable set of propositional variables.

Informally, the semantics of the temporal operators is the following (for other operands, see [15]): the A and E mean, *inevitably in all paths* and *possibly in a path*, respectively. The rest are X, F or U, meaning *next state*, *eventually in some state* and *until*, respectively. For further details about the syntactic properties over the CTL formulas, see [15]. In Example 2.1, we show how CTL is more expressive than the propositional logic through a formalization of the mutual exclusion property.

Example 2.1. *The mutual exclusion property, Figure 2.1, is announced as follows [12]:*

The process C_1 and process C_2 are never in a critical region at the same time.

A possible propositional logic formula of the property could be:

$$\neg p \wedge \neg q \tag{2.2}$$

where p means "process C_1 is in a critical region", and q means "process C_2 is in a critical region".

If we consider the model where p is false, then Formula 2.2 is satisfiable. However, we are forgetting the essence of the mutual exclusion property: the property needs to be true all the time. Using the syntax and informal semantics of CTL, a better formalization is the following:

$$\neg EF\ (p \wedge q) \tag{2.3}$$

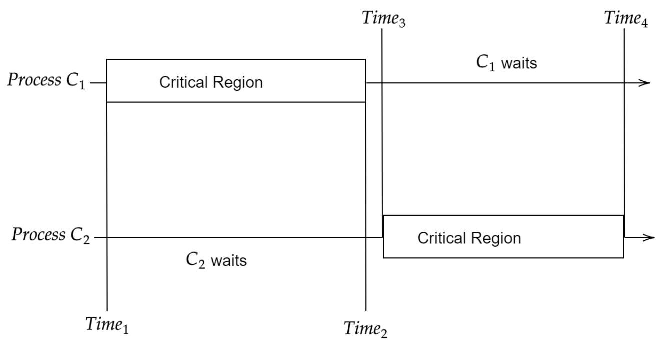

Figure 2.1: Mutual exclusion property for two processes.

In order to do model checking over CTL, we need a mathematical structure that models the analyzed software system. In our particular case, we use Kripke structures:

Definition 2.3 (Kripke Structure [9]). A Kripke structure K is a tuple (S, I, R, L) where:

- S is a finite set of *states.*
- $I \subseteq S$, called *set of initial states.*
- $R \subseteq S \times S$ and R is total relation, where a total relation is defined as: for every $s \in S$ there is $s' \in S$ such that $(s, s') \in R$.
- $L : S \rightarrow 2^P$, where P is a set of propositional variables. This function is called *labelling function* and identifies the propositional variables that are true in a specific state of K.

The AF and $E[\_\ U\ \_]$ operators use a path interpretation over the Kripke structures. A path is simply an infinite succession of states $s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R$ for $i \in \mathbb{N}$. As is usual, we denote the paths as states separated by arrows: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$.

Now, we can define a relation of satisfiability between a Kripke structure K, a state of K and a CTL formula.

Definition 2.4 (Satisfiability [9]). Let $K = (S, I, R, L)$ be a Kripke Structure, $s \in S$ and φ be a formula. We define $K, s \models \varphi$ inductively over the structure of φ:

- $K, s \models v$ where v is a propositional variable iff $v \in L(s)$.
- $K, s \models \neg\psi$ iff $K, s \not\models \psi$.
- $K, s \models \psi_1 \wedge \psi_2$ iff $K, s \models \psi_1$ and $K, s \models \psi_2$.
- $K, s \models AX\ \psi$ iff for all $s' \in S$, if $(s, s') \in R$ then $K, s' \models \psi$.
- $K, s \models EX\ \psi$ iff exists $s' \in S$ such that $(s, s') \in R$ and $K, s' \models \psi$.
- $K, s \models EF\ \psi$ iff exists a path $s_0 \to s_1 \to s_2 \to \dots$ where $s = s_0$, and exists an s_i such that $K, s_i \models \psi$.
- $K, s \models AF\ \psi$ iff for all paths $s_0 \to s_1 \to s_2 \to \dots$ where $s = s_0$, there exists an s_i such that $K, s_i \models \psi$.
- $K, s \models E[\psi_1\ U\ \psi_2]$ iff there exists a path $\pi = s_0 \to s_1 \to s_2 \to \dots$ where $s = s_0$, and exists an s_i in the path π such that $K, s_i \models \psi_2$, and for each $j < i$, we have $K, s_j \models \psi_1$.
- $K, s \models A[\psi_1\ U\ \psi_2]$ iff for all paths $\pi = s_0 \to s_1 \to s_2 \to \dots$ where $s = s_0$, there exists an s_i in the path π such that $K, s_i \models \psi_2$, and for each $j < i$, we have $K, s_j \models \psi_1$.

We observe that Definition 2.4 establishes the relation of satisfiability over a single state s. In the case of program verification, the intuition could be that s is an input of the program. Thus, in order to model check, we need to quantify the states over the set of initial states or all the possible inputs of the program [7, 15]; see Definition 2.5.

Definition 2.5. Let $K = (S, I, R, L)$ be a Kripke structure and φ be a CTL formula. We define model checking for CTL as the procedure to verify if the following holds:

$$\forall s \in I.\ K, s \models \varphi \tag{2.4}$$

At this point, we want to automatically verify Equation 2.4. In the next section, we introduce an algorithm that achieves this goal. This algorithm, called *labelling algorithm* [15], solves the model checking problem for CTL.

2.3 A labelling algorithm

A naive way of doing model-checking for CTL in a computer is to implement Definition 2.4. The program verifies if $K, s_0 \models \varphi$ holds by taking as inputs a Kripke structure K, an initial state s_0 of K, and a CTL formula φ. However, a direct implementation of Definition 2.4 cannot check φ

over the infinite paths created by the unwinding of K, as the computer only checks finite data structures. [15]

Another strategy we can use is to compute the subset of states S' such that for every state s' in S', $K, s' \models \varphi$ holds. This approach gives us the benefit of analyzing some states only when it is necessary. Thus, we can verify if the initial state(s) belong(s) to S'. This idea is the essence of the *labelling algorithm* [15].

As other logic procedures, we compact the grammar of Equation 2.1 into the grammar of Equation 2.7. In [15], Huth & Ryan establish that this reduction has given good results in the practice.

In order to achieve the grammar reduction, we use the equivalences of Equation 2.5 and assume the symbol $\top$ as the formula $p \vee \neg p$ for any propositional variable p.

$$AX\ \varphi \equiv \neg EX\ \neg\varphi \tag{2.5}$$

$$EF\ \varphi \equiv E[\top\ U\ \varphi] \tag{2.6}$$

$$\varphi\ ::=\ v \mid \neg\varphi \mid \varphi \wedge \varphi \mid EX\ \varphi \mid E[\varphi\ U\ \varphi] \mid AF\ \varphi \tag{2.7}$$

At this point, we split the formulas into two categories of operators: propositional and temporal. With the last categorization, we search for the states that satisfy the propositional formulas by giving a set interpretation of the propositional operators (see Table 2.1).

Operator	Set interpretation
$\neg$	Complement
$\wedge$	Intersection ($\cap$)
$\vee$	Union ($\cup$)

Table 2.1: A set interpretation of the propositional operators

We take the following strategy for the temporal operators: we search for a state where the variable p is true and then *go backwards* in order to satisfy the temporal semantics of the operator. Let us begin with the EX operator.

The EX operator is the simplest one. Let us suppose that we have the states s_0, s_1, the labels $L(s_0) = \{p, r\}$, $L(s_1) = \{q, r, t\}$ and the following path:

$$s_0 \rightarrow s_1 \rightarrow \ldots$$

We know that q is true in s_1 and s_1 is the *next* state of s_0. Therefore, the state s_0 satisfies $EX\ q$.

For the rest of the operators, we *rewrite* each one in terms of itself, as we show in the following equations:

$$E[\varphi_1\ U\ \varphi_2] \equiv \varphi_2 \vee (\varphi_1 \wedge EX(E[\varphi_1\ U\ \varphi_2])) \tag{2.8}$$

$$AF\ \varphi \equiv \varphi \vee AX(AF\ \varphi) \tag{2.9}$$

As we establish in Table 2.1, we can give a set interpretation of Equations 2.8 and 2.9. Also, we use the EX strategy to search for the states that satisfy $E[\_U\_]$ and AF.

The main difference between $E[\_U\_]$ and AF is the prefix. In the case of E, we search for *one* state for the satisfaction of the formula. By contrast, in the case of A, we search for *all* the initial states that satisfy the rest of the formula. We have two possibilities to deal with AF: One possibility would be to satisfy the universal condition. Another possibility would be to express the universal condition in terms of an existential condition. We take the first option.

We present the main procedure in Algorithm 1, the EX strategy in Algorithm 2, the $E[\_U\_]$ strategy in Algorithm 3 and, the AF strategy in Algorithm 4. In order to give a more elegant version of the labelling algorithm, our presentation of the labelling algorithm differs in some notations from the algorithm presented by Huth & Ryan in [15]. We use a combination of *Haskell-C-pseudocode* style.

Algorithm 1 Labelling algorithm

```
Require: CTL formula $\varphi$, Kripke Structure $K = (S, s_0, R, L)$
Ensure: List of states that satisfies $\varphi$
procedure LABELLING($\varphi$,$K$)
    case ($\varphi$) of
    $p \Rightarrow$ return $\{s \in S \mid p \in L(s)\}$
    $\neg\varphi \Rightarrow$ return $S-$LABELLING($\varphi$,$K$)
    $\varphi_1 \wedge \varphi_2 \Rightarrow$ return LABELLING($\varphi_1$,$K$)$\cap$LABELLING($\varphi_2$,$K$)
    $EX\ \varphi \Rightarrow$ return EXCASE($\varphi$)
    $E[\varphi_1\ U\ \varphi_2] \Rightarrow$ return EUCASE($\varphi_1$,$\varphi_2$)
    $AF\ \varphi \Rightarrow$ return AFCASE($\varphi$)
    end case
end procedure
```

Apparently, the labelling algorithm is our best solution for the model-checking problem. However, the size of the Kripke structures is more often than not exponential in the number of variables. This tendency over the size of the Kripke structures is a problem called the *state-explosion problem*, and affects the performance of the labelling algorithm.

Algorithm 2 Case of EX operator

Require: CTL formula φ, Kripke Structure $K = (S, s_0, R, L)$

Ensure: List of states that satisfies $EX\ \varphi$

```
1: function EXCase($\varphi$,$K$)
2:     $X \leftarrow$Labelling($\varphi$,$K$)
3:     $Y \leftarrow \{s \in S \mid \exists s' \in S, (s, s') \in R \land s' \in X\}$
4:     return Y
5: end function
```

Algorithm 3 Case of $E[\_U\_]$ operator

Require: CTL formula φ, Kripke Structure $K = (S, s_0, R, L)$

Ensure: List of states that satisfies $E[\varphi_1 U \varphi_2]$

```
1: function EUCase($\varphi$,$K$)
2:     $W \leftarrow$Labelling($\varphi_1$,$K$)
3:     $X \leftarrow S$
4:     $Y \leftarrow$Labelling($\varphi_2$,$K$)
5:     while $X \neq Y$ do
6:         $X \leftarrow Y$
7:         $Y \leftarrow Y \cup (W \cap \{s \in S \mid \exists s' \in S, (s, s') \in R \land s' \in Y\})$
8:     end while
9: end function
```

Algorithm 4 Case of *AF* operator

```
Require: CTL formula $\varphi$, Kripke Structure $K = (S, s_0, R, L)$
Ensure: List of states that satisfies $AF\ \varphi$
1: function AFCase($\varphi$,$K$)
2:     $X \leftarrow S$
3:     $Y \leftarrow$Labelling($\varphi$,$K$)
4:     while $X \neq Y$ do
5:         $X \leftarrow Y$
6:         $Y \leftarrow Y \cup \{s \in S \mid \forall s' \in S, (s, s') \in R \rightarrow s' \in (Y)\}$
7:     end while
8: end function
```

In the next section, we introduce a well-known data structure, called *Ordered Binary Decision Diagram* (OBDD), that *symbolically* represents a set of states. Set manipulation over this *symbolic representation* is efficient. If we implement the set of states manipulated by the labelling algorithm with this data-structure, the labelling algorithm may be able to overcome the state-explosion problem [15].

2.4 OBDDs: from enumerative to symbolic checking

Symbolic methods use boolean formulas as symbols and have been effective for solving large instances of the model-checking problem [9]. When such methods represent sets of states with symbols, it is possible to readily incorporate them in the labelling algorithm. As a result, we will get a better performance over other implementations of the labelling algorithm [15].

A data structure for representing and manipulating boolean formulas in a symbolic form is the *Ordered Binary Decision Diagram* (OBDD). An OBDD is a directed acyclic graph where the variables that occur in the graph are ordered [9]. As we said, one of the basis of the labelling algorithm is the set manipulation. Implementing the labelling algorithm set manipulation as OBDDs operations results in the desired symbolic strategy [15].

First, let us formally define a BDD, and then give an order over the variables that occur in the BDDs resulting in OBDDs.

Definition 2.6 (Binary Decision Diagram)**.** A Binary Decision Diagram (BDD) is a directed acyclic graph with a unique initial node, where:

- The terminal nodes are labeled with either 0 (the false constant) or 1 (the true constant).

- The non-terminal nodes are labelled with a boolean variable and they only have two labelled edges.
- Each edge is labelled with either 0 or 1.

Definition 2.7 (Ordered Binary Decision Diagram)**.** Let $X = [x_1, \ldots, x_n]$ be an ordered list of boolean variables without duplications, and B be a BDD where all the variables of X appear in B. We say that B has the ordering X if all variable labels of B occur in X, and for every $x_i, x_j \in X$ such that x_i is followed by x_j along any path in B, we have $i < j$.

We are assuming that all the OBDD's that we use in this this thesis are reduced.

There are various references on how to operate the OBDDs [9, 10, 15]. In this thesis, we focus on how to use the OBDDs in order to solve the model-checking problem. If we want to implement the labelling algorithm set manipulations as OBDDs operations, we will need to represent the Kripke structure and the CTL formula as OBDDs.

We begin by representing a Kripke structure as an OBDD. We use the definitions provided by Déharbe in [10] and Huth & Ryan in [15]. First, we show how to construct the characteristic function of a subset of states of a Kripke structure.

Definition 2.8 (State representation)**.** Let $K = (S, R, I, L)$ be Kripke structure and, $V = \{v_1, \ldots, v_n\}$ be the set of variables that occur in K. We denote by v any vector that contains all the variables of V, i.e., $v = (v_1, \ldots, v_n)$. The characteristic function of a state s is defined as:

$$[s](v) = (\bigwedge_{v_i \in L(s)} v_i) \cdot (\bigwedge_{v_i \notin L(s)} \overline{v_i}) \tag{2.10}$$

where $\bigwedge_{x \in L(s)} x_i = x_1 \cdot x_2 \cdot \cdots \cdot x_n$.

The generalization of Definition 2.8, to set of states, is simple. We recursively compute the characteristic function of each state s and compose each characteristic function with a disjuction $(+)$. For simplicity, we overload the brackets.

Definition 2.9 (States representation)**.** Let $K = (S, R, I, L)$ be a Kripke structure, $V = \{v_1, \ldots, v_n\}$ be the variables that occur in K and, $X \subseteq S$. The characteristic function of X is defined as:

$$[\emptyset](v) = 0 \tag{2.11}$$

$$[\{s\} \cup X](v) = [s](v) + [X](v) \tag{2.12}$$

The idea of how to define the characteristic function of the relation of a Kripke structure is similar to that of the state representation. First, we define the characteristic function of a transition:

Definition 2.10 (Transition representation)**.** Let $K = (S, R, I, L)$ be a Kripke structure, $V = \{v_1, \ldots, v_n\}$ be the variables that occurs in K and, $V' = \{v'_1, \ldots, v'_n\}$ be a primed copy of V. The characteristic function of a transition between two states is defined as:

$$[(s_1, s_2)](v, v') = [s_1](v) \cdot [s_2](v') \quad \text{iff} \quad (s_1, s_2) \in R$$

The characteristic function of the relation of a Kripke structure is analogous to the subset of states representation.

Definition 2.11 (Relation representation)**.** Let $K = (S, R, I, L)$ be a Kripke structure, $V = \{v_1, \ldots, v_n\}$ be the variables that occur in K and, $V' = \{v'_1, \ldots, v'_n\}$ be a primed copy of V. The characteristic function of R is defined as:

$$[\emptyset](v, v') = 0 \tag{2.13}$$

$$[\{(s_1, s_2)\} \cup R](v, v') = [(s_1, s_2)](v, v') + [R](v, v') \tag{2.14}$$

The case of CTL formulas is simple. In the previous section, we redefine the temporal operators in terms of themselves by using propositional connectives. Thus, we can construct the OBDD for the boolean formulas as Huth & Ryan do in [15].

We denote f as the characteristic function of the OBDD that represents B_f. In Table 2.2, we show the boolean formulas and their OBDD representation. We denote by $f[x \mapsto b]$ the replacement of all the occurrences of variable x by the boolean constant b in f.

The last step we need is to represent the sets (called preimages) computed by the Algorithms 2, 3 and, 4. We represent the preimages as characteristic functions [10]:

$$\{s \in S \mid \exists s' \in S, (s, s') \in R \land s' \in Y\} \rightsquigarrow \exists v', [\![R]\!](v, v') \cdot [\![Y]\!](v') \tag{2.15}$$

$$\{s \in S \mid \forall s' \in S, (s, s') \in R \rightarrow s' \in Y\} \rightsquigarrow \forall v', \overline{[\![R]\!](v, v')} + [\![Y]\!](v') \tag{2.16}$$

With the previous transformations, we replace the propositional connectives and the sets by OBDDs that represent the characteristic functions [10].

This new version of the labelling algorithm is more efficient, for some real cases, than the version we showed in the previous section [15]. However, the labelling algorithm has the characteristic of being more or less efficient, depending on how we *order* the variables of the variables that occur in the OBDDs. In the next subsection, we show how the order of the variables affects the size of the OBDDs and consequently, the complexity of the OBDD operations.

2.4.1 To sort or not to sort? The OBDD NP-hard question

Various authors establish the relationship between the order of the OBDDs and the efficiency of the labelling algorithm [9, 10, 15]. Let us introduce how the operations affect the complexity of

Boolean formula f	OBDD B_f
1	Leaf 1
0	Leaf 0
p	$Node(p, 0, 1)$
$\overline{f}$	Swap the 0 and 1 nodes in B_f
$f + g$	$\texttt{apply}(+, B_f, B_g)$
$f \cdot g$	$\texttt{apply}(\cdot, B_f, B_g)$
$f[x \mapsto 0]$	$\texttt{restrict}(0, x, B_f)$
$f[x \mapsto 1]$	$\texttt{restrict}(1, x, B_f)$
$\exists x.f$	$\texttt{apply}(+, B_{f[x\mapsto 0]}, B_{f[x\mapsto 1]})$
$\forall x.f$	$\texttt{apply}(\cdot, B_{f[x\mapsto 0]}, B_{f[x\mapsto 1]})$

Table 2.2: OBDD representation of the characteristic functions.

the labelling algorithm; for further details see [9, 10, 15].

If we want to analyze the complexity of the OBDD operations, we need to define the size of the OBDDs.

Definition 2.12 (OBDD size)**.** Let B be an OBDD. The size of the OBDD, denoted as $|B|$, is the number of nodes that occur in B.

According to Huth & Ryan in [15], the time complexity of the OBDD operations have the upper bounds that we show in Table 2.3.

Operation	Time complexity
`reduce`	$O(\|B\| \cdot \log(\|B\|))$
`apply`	$O(\|B_f\| \cdot \|B_g\|)$
`restrict`	$O(\|B_f\| \cdot \log(\|B_f\|))$

Table 2.3: Complexity of the OBDD operations. B_f and B_g are the OBDD representations of the characteristic functions f and g. B is the OBDD given as input.

It is clear how the OBDD operations could affect the complexity of the labelling algorithm.

Example 2.2. *Consider the characteristic function for the n-bit comparator:*

$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = \bigwedge_{i=1}^{n} (x_i \leftrightarrow y_n) \tag{2.17}$$

If we consider two different orders, then the number of nodes could either be polynomial or exponential [9]:

Order	Number of nodes
$[x_1, y_1, x_2, y_2, \ldots, x_n, y_n]$	$3n + 2$
$[x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n]$	$3 \cdot (2^n - 1)$

Table 2.4: Two orders for the n-bit comparator.

If we choose the second order of Table 2.4, the complexity of any OBDD operation will be absorbed by the exponential number size of the second OBDD.

We could try to define an algorithm to search for the best order for the OBDD. This search has been proved to be NP-hard and more than that, sometimes it is impossible to find a good order over the OBDD, i.e., there exist not sortable OBDD's [3]. There are some heuristics for real-life problems that have shown this is not an inconvenient. In the particular case of this thesis, we create an ordering that showed an improvement in our model-checking problem.

In the next section, we mention two state-of-the-art model checkers and some practice guarantees of the model checker we have use in this thesis: nuXmv.

2.5 nuXmv and its guarantees

We are interested in a software that is an implementation of a model checking procedure. In the particular case of temporal logics, these software solutions are called *model checkers* (see Figure 2.2).

Some examples of symbolic model checkers are:

- NuSMV: Is a symbolic model checker that implements the BDD and SAT-based symbolic model checking. NuSMV is reimplementation and extension of the model checker SMV [6]. Official link: `http://nusmv.fbk.eu/`
- nuXmv: Is an extension of NuSMV with the capacity to work with infinite-sate models for the LTL case [5]. Official link: `https://nuxmv.fbk.eu/`

nuXmv is a state-of-the-art model checker, and their responses (especially the counterexamples) are adequate for our main contribution: *reCipe*. Also, nuXmv participated in the *Hardware Model*

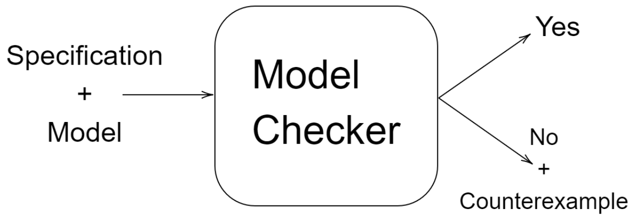

Figure 2.2: Concept of a model checker

Checking Competition (2013[1] and 2014[2]) being in the three first places. Therefore, we think it is a good option to deal with our problem.

[1]http://fmv.jku.at/hwmcc13/

[2]http://fmv.jku.at/hwmcc14/

Chapter 3

Temporal Logic Guided Inductive Synthesis

"We expect synthesis algorithms to provide us with some control over the space of programs that are going to be considered, not just their intended behavior."

Solar-Lezama, Armando (2018) [24]

Our main contribution is the development of *reCipe*, a new program-synthesis algorithm that relies on model-checking for CTL. reCipe is a synthesis procedure that generates candidate expressions by providing a richer specification against more traditional sketch-based synthesis approaches that lets the user describe how values evolve over time.

In this chapter, we describe reCipe. We begin with a cartoon version of reCipe, then we present an illustrative example of how reCipe works, and we end with a formalization of the algorithm and a proof of its correctness.

3.1 A cartoon version

Let us remember that the *program-synthesis* problem consists in generating a program P such that for all possible inputs, P satisfies a user specification φ. We can formalize the program-synthesis problem as a predicate $Q(p, x)$ where p denotes a program, x denotes the inputs, and the meaning of Q is the desired program behavior. Thus, the program-synthesis problem reduces to finding a program behavior as the following equation establishes:

$$\exists p. \forall x. Q(p, x) \tag{3.1}$$

On the one hand, Solar-Lezama in [23] mentions that solving equations as Equation 3.1 is PSPACE-complete [11]. On the other hand, we can handle the model-checking problem for models with a vast number of states [9]. Therefore, we can obtain the benefits of the model-checking techniques by translating Equation 3.1 into a CTL formula.

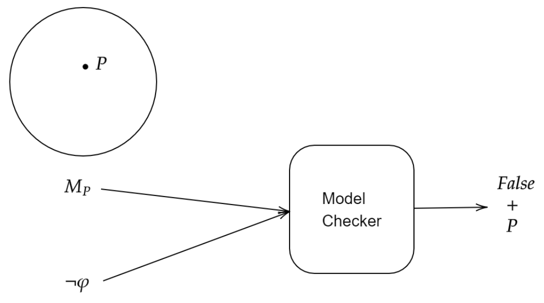

Figure 3.1: From model-checking to program-synthesis

As we show in Figure 3.1, the main idea of the translation is simple: assume that a model M_P has program traces Π. One trace in Π has a subtrace P such that P satisfies the user specification φ. Therefore, if we input $M_P \models \neg\varphi$ into a model checker, such tool will return *False* and as a counterexample, the trace that has P.

The previous argument is the prior work of our contribution [19]. Despite the simplicity of our proposal, there are some challenges: How to encode the synthesis procedure and the program traces into M_P and how to transform the user specifications in such way that *forces* a model checker to return P. In order to answer the previous questions, in the next section we show how reCipe works by giving an illustrative example.

3.2 Illustrative Example

Consider a program with the following behavior: "given two initialized integer variables, $x = v_1$ and $y = v_2$, the program swaps the values respectively stored in the variables x and y, i.e., $x = v_2$ and $y = v_1$." The goal of this programming task is to create a program by only using x and y.

The first step is to formalize the specification into a CTL formula with some syntactic sugar. In our example, Formula 3.2 formalizes the above specification.

$$\forall n.\forall m.(x = n \land y = m \to EF\,(x = m \land y = n)) \tag{3.2}$$

In case the formula has first-order quantifiers, we unfold the quantifiers by using a finite set of integers $\mathcal{D}$. In our example, 3.3 is the quantifier-free formula of Equation 3.2.

$$\bigwedge_{n_i, m_j \in \mathcal{D}} (x = n_i \land y = m_j \to EF\,(x = m_j \land y = n_i)) \tag{3.3}$$

Since we only have Equation 3.3, the next step is challenging: create a program structure. If we start from the variables and values that occur in a CTL formula φ, we can create a generic program structure S. If we define a generic program structure as a template of a program, the *sketch* [22] is a good representation of the program structure. The construction of the sketch is simple: using *go-to* commands for each variable of φ, we create a non-deterministic choice between an `if`-conditional and a while loop. For example, with the variables x and y that occur in Formula 3.3, we construct

```
void func(int* x,int* y){
  l1:if(*){goto l2;}else{goto l3;}
  l2:*x = ??;goto {l1,l3};
  l3:if(*){goto l4;}else{goto l5;}
  l4:*y = ??;goto {l3,l5};
  l5:if(*){goto l6;}else{goto l7;}
  l6:*x = ??;goto {l5,l7};
  l7:if(*){goto l8;}else{goto l9;}
  l8:*y = ??;goto {l7,l9};
  l9:return;
}
```

Sketch 3.1: The program structure generated from the Formula 3.3.

a code with "holes", Sketch 3.1. In Section 3.5, we give a more formal construction of these generic sketches.

We can get a program from Sketch 3.1 by filling in the holes and removing the non-determinism. In other words, we *search* for the expressions that complete our sketch. If we model this search as a search tree, then at the first level, we only have a sketch, and at the next level, each node has a set of candidate expressions. Finally, in the last level, we *take* each candidate, complete the sketch and unfold the behavior of the program with every input. Using this search tree, we can verify the user specification at the last level of the tree and get the desired program.

We encode the last procedure as a Kripke structure K. As Clarke et al. in [9], we define the states of K as propositions of the form $(variable = value)$. The main difference between our Kripke structures (synthesis procedure) and Clarke's Kripke structures (program behavior) is that whereas we need three categories of variables, Clarke's Kripke structures need only the set of the program variables and the variable that represent the program flow. In our case, we need variables that represent the synthesis steps, variables that represent the holes that will be filled, and variables that represent the program behavior.

We model the search tree as the Kripke structure in Figure 3.2. The *parts* of the Kripke structure *divide* the searching steps (we call them *synthesis steps*):

- Step q_0: We only have the sketch.
- Step q_{sel}: We generate a variety of expressions that complete the sketch.

- Step q_1: We choose the candidates of q_{sel} step, complete the sketch, and execute the completed sketch with every input of domain D.

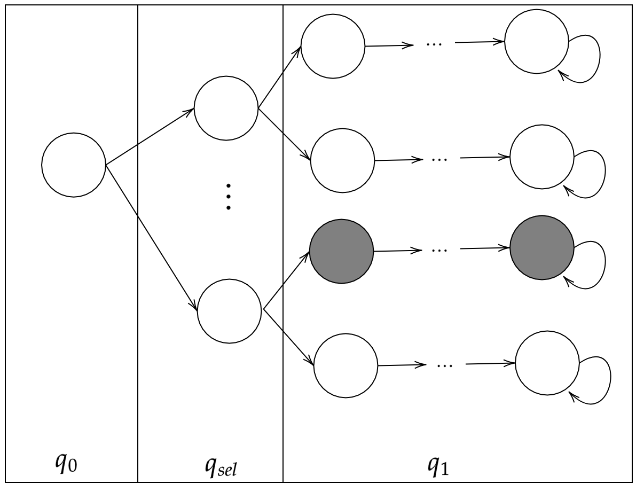

Figure 3.2: Kripke structure with synthesis states.

To represent the process of completing a sketch and to analyze the resulting program, we need variables whose value is an abstract representation of an expression; these values will be static after the q_{sel} step. Therefore, we replace the holes of a sketch (`??`) and the non-deterministic variables, with new variables exp_i, where $i \in \{1, \ldots, n\}$ and n is the number of holes. In Code 3.2 we have replaced the variables in order to generate a program.

In the synthesis step q_1, we need to complete the sketch and then try the resulting program with every input of $\mathcal{D}$. For the first part, assume that a trace π has the gray colored states of Figure 3.2. In order to complete a sketch, we use the values of each exp_i stored in π. In our example, we show the resulting program in Code 3.1.

The last ingredient we need is a CTL formula with a similar interpretation as Equation 3.1. In Table 3.1, we show the temporal operators, the transition between the synthesis states when we apply the semantics, and the response we expect. The semantics of the CTL operators allows us to construct the formula, formalizing the process of synthesis.

Temporal operator	Transition between synthesis states	Response
EX	$q_0 \Rightarrow q_{sel}$	A sample of candidate expressions
AX	$q_{sel} \Rightarrow q_1$	The behavior of the program

Table 3.1: Temporal operators for reCipe formula

```
void func(int* x,int* y){
  l1:if(*){goto l2;}else{goto l3;}
  l2:*x = exp1;goto {l1,l3};
  l3:if(*){goto l4;}else{goto l5;}
  l4:*y = exp2;goto {l3,l5};
  l5:if(*){goto l6;}else{goto l7;}
  l6:*x = exp3;goto {l5,l7};
  l7:if(*){goto l8;}else{goto l9;}
  l8:*y = exp4;goto {l7,l9};
  l9:return;
}
```

Sketch 3.2: Sketch with named holes.

If we add the previous operators and the negation to the formalized user specification, we will obtain the desired trace as a counterexample. In our example, we input Formula 3.3 and the Kripke structure of Figure 3.2 into a model checker, getting the desired values as we show in Response 3.1.

$$\neg EX\,AX\,(\bigwedge_{n_i,m_j\in\mathcal{D}}(x=n_i\wedge y=m_j\rightarrow EF\,(x=m_j\wedge y=n_i))) \tag{3.4}$$

```
Irrelevant information, state 1.2 is at the synthesis step qsel
--> State 1.2 <--
exp0 = plvxvy
exp1 = mvxvy
exp2 = mvxvy
exp3 = vy
```

Response 3.1: Counterexample path

The `State 1.2` in Response 3.1 gives us a counterexample for the AX operator of Formula 3.4. Using a similar strategy, we can synthesize the guards and remove the non-determinism of the *go-to* statements. In our example, we obtain Code 3.1.

```
void func(int* x,int* y){
  l1:if(*x !=*x + *y){goto l2;}else{goto l3;}
  l2:*x = *x + *y;goto l1;
  l3:if(*x !=*x + *y){goto l4;}else{goto l5;}
  l4:*y = *x - *y;goto l3;
  l5:if(*x !=*x + *y){goto l6;}else{goto l7;}
  l6:*x = *x - *y;goto l7;
  l7:if(true){goto l7;}else{goto l9;}
  l8:*y = *y;goto l9;
  l9:return;
}
```

Code 3.1: The swap algorithm

The shown procedure is straightforward, but there are some questions: How to create and use a Kripke structure in order to encode the synthesis procedure for any program. Before we discuss the solution to these challenges, in the next section, we show the pseudocode of reCipe.

3.3 reCipe algorithm

In this section, we present the pseudocode of *reCipe*. Given a user specification φ and a number of iterations, reCipe returns a list of symbolic expressions for completing a sketch that satisfies φ. In case no program satisfies φ, reCipe returns an empty list.

The main procedure (Algorithm 5) computes an input domain $\mathcal{D}$, then the algorithm transforms φ in a pure CTL formula using $\mathcal{D}$. Finally, reCipe recursively creates a set of expressions that satisfies φ.

At the first step, we execute *bounds*, a function that computes a data domain. For example, if the user specification only has integers, *bounds* computes $\mathcal{D} = \{minC, \ldots, maxC + 4\}$ such that $minC$ is the minimum value and $maxC$ is the maximum value that appears in φ. If the user specification does not have integers, *bounds* returns $\mathcal{D} = \{0, \ldots, 4\}$. The constants zero and four were obtained in an empirical way.

The next step of reCipe is to execute the function *newForm* uses the domain $\mathcal{D}$ in order to unfold the quantifiers that occur in φ, adds the prefix $\neg EX\ AX$ to the unfolded formula and stores the result in ψ. Finally, the algorithm calls the recursive subroutine reCipe_AUX (Algorithm 6) with ψ, the number of iterations n and a parameter that indicates the depth of the abstract syntax tree.

Algorithm 5 reCipe Algorithm

```
Require: User specification $\varphi$, number of iterations $n$
Ensure: List of symbolic expressions that satisfies $\varphi$ or error
1: procedure reCipe($\varphi$,$n$)
2:     $(minC, maxC) \leftarrow bounds(\varphi)$
3:     $\psi \leftarrow newForm(\varphi, minC, maxC + 4)$
4:     $\mathcal{P} \leftarrow reCipe\_aux(\psi, n, 1)$
5: end procedure
6: Return: $\mathcal{P}$
```

The subroutine reCipe_AUX (Algorithm 6) is a recursive procedure that creates a Kripke structure K with some candidate expressions whose abstract syntax tree has height i. If $K \not\models \psi$, a counterexample is generated, ending the subroutine. Otherwise, reCipe_AUX is called recursively increasing i by one, i.e., we try with more complex programs because in the current iteration we could not find a program. This process ends when we either get a counterexample or there does not exist a program that satisfies the user specification.

The *getTrace* function calls the model checker nuXmv. If nuXmv finds a counterexample, nuXmv returns a state of K where the desired expressions complete the generic sketch. Otherwise, the algorithm makes a recursive call where the algorithm creates more complicated expressions by adding a deeper level in the bottom-up search.

The algorithm ends with one of the following scenarios:

1. The algorithm finds a counterexample trace.
2. The algorithm does not find a counterexample.

There are two main reasons for the second case: first, the user specification is not exact or the user specification is wrong; second, the expressions need to be more complicated in order to satisfy the user specification.

In the next section, we show the correctness of reCipe by proving properties over the Kripke structure constructed by our algorithm.

Algorithm 6 reCipe_AUX

Require: User specification without quantifiers ψ, number of iterations n, current iteration i

```
1: procedure reCipe_aux($\psi$,$n$,$i$)
2:     $\mathcal{D} \leftarrow [minC..maxC]$
3:     $\mathcal{V} \leftarrow vars(\psi)$
4:     $\mathcal{M} \leftarrow constructKripke(\mathcal{D}, \mathcal{V}, i)$
5:     if $K \not\models \psi \wedge i \leq n$ then
6:         $\mathcal{P} \leftarrow getTrace(\mathcal{M}, \psi)$
7:     else
8:         $reCipe\_aux(\psi, n, i + 1)$
9:     end if
10: end procedure
```

3.4 Correctness of reCipe

To synthesize a program, Morgenstern and Schneider in [19] assume that the user gives a sketch S, see [23], and their procedure fills in S by using the procedure we show below. Our contribution has two main differences compared with the work of Morgenstern and Schneider. First, we construct S by extracting information from a user specification. Second, we remove the non-determinism caused by the election between a while cycle or an `if`-statement. In the following, we show the non-determinism removal and in the next section, we show the sketch generation.

Assume that we used the information of a user specification φ and constructed Sketch 3.3. In order to satisfy φ, we need to replace each $\mathtt{e_i}$, $\mathtt{g_i}$ and non-deterministic `goto` statement by an expression, a predicate and a deterministic `goto` statement. Let us show how to achieve this replacement.

In the rest of this section, we formally define the candidates we mentioned in the previous sections. In other words, we define the candidates that will replace the holes of Sketch 3.3.

Definition 3.1 (Candidate)**.** Let c be a sequence of expressions, guards and deterministic `goto`-statements:

$$[e_1, \ldots, e_n, g_1, \ldots, g_n, gt_1, \ldots, gt_n]$$

where n is the number of variables that occur in Sketch 3.3 and each e_i, g_i, gt_i is an expression, a guard or a deterministic `goto`-statement, respectively.

```
void func(type x1,...,type xn){
    l1: if(g1){goto l2;}else{goto l3;}
    l2: x1 = e1;goto {l1,l3};
    ...
    ln_1: if(gn){goto ln;}else{goto lr;}
    ln: xn = en;goto {ln_1,lr};
    lr:return;
}
```

Sketch 3.3: One-iteration sketch generated by reCipe

It is easy to define a function that takes a sketch and a candidate, and returns a program. We only need to replace each hole of the sketch by an expression, a guard or a deterministic `goto`-statement, according to each case. Let us call this function F.

Once we have a procedure to fill in the holes in Sketch 3.3, we want to represent a program behavior as a Kripke structure. Clarke et al. in [9] give a translation of a program behavior, with an input i, to a Kripke structure. Let us denote this translation as the function C. Thus, we can get a Kripke structure by computing $C(F(S_p, c), i)$ where S_p is a sketch, c is a candidate and i is an input. For further details of how C is defined, see [9].

Clarke et al. in [9] define the elements of Kripke structures as first-order formulas, where the set of states and transitions are valuations that satisfy their respectively formulas. We use the set notation in order to simplify our notation. For example, if the initial state is defined by the formula $v = x \wedge \bigwedge_{i=1}^{n} \neg(v = y_i)$, we define the initial state as the set $v = x$. In other words, the valuations that do not appear in a state or as a part of a pair of the relation are false.

Let S_p be Sketch 3.3, c be a candidate and i be an input, then the resulting Kripke structure K_c^i of $C(F(P, c), i)$ has the following form:

$$K_i^c = (S, S_0, R, L) \tag{3.5}$$
$$S = \{v = i \mid v \in \mathcal{V}, i \in \mathcal{D}\} \tag{3.6}$$
$$S_0 \subseteq S \tag{3.7}$$
$$R = \{(s, s') \mid \forall v \in \mathcal{V}.\forall v' \in \mathcal{V}'.(s(v), s'(v')) \models \mathcal{R}\} \tag{3.8}$$
$$L(s) = \{v \mid s(v) = True\} \tag{3.9}$$

where $\mathcal{R}$ is the formula that represents the transitions of K_i^c.

Since we have a different program P for each candidate, and a different behavior of P for an input i, we have a Kripke structure K_i^c for each input i and a fixed c, as we show in Figure 3.3.

Analogously, we have different programs for different candidates. For example, in Figure 3.3 each Kripke structure K_i^c is defined by a different candidate c_i.

With the last construction, we have defined all the possible programs and their behaviors for each possible input. The next step to encode is the candidate election. If we "wire" the initial state of each K_i^c, for a fixed candidate c, then we have encoded the election of candidate c. We repeat this strategy with every candidate c, as we show in Figure 3.3.

The last step is the beginning of reCipe. We wire the previously constructed Kripke structures with a single initial state. This initial state represents where we begin to fill in the sketch. We show a diagram of this final Kripke structure in Figure 3.3.

Let us formally define the Kripke structure of Figure 3.3.

Definition 3.2. Let S_p be Sketch 3.3, n be the number of variables that occur in S_p, and cs be a set of candidates $\{c_1, \ldots, c_n\}$. First, let us define $K_i^{c_i} = (S_{c_i}, S_{0c_i}, R_{c_i}, L_{c_i})$ as the Kripke structure that encodes the program behavior of the completed sketch S_p with the candidate c and input i. Now, we define the Kripke structure K_S that encodes the process of completing S as follows:

$$K_{S_p} = (S, S_0, R, L) \tag{3.10}$$

$$S = \mathit{Init} \times \mathit{Candidates} \times \bigcup_{i=1}^{n} S_{c_i} \tag{3.11}$$

$$\mathit{Init} = \{\mathit{init} = \mathit{true}\} \tag{3.12}$$

$$\mathit{Candidates} = \{c = c_i \mid c_i \in cs\} \tag{3.13}$$

$$S_0 = \{\mathit{init} = \mathit{true}\} \tag{3.14}$$

$$R = \{(\mathit{init} = \mathit{true}, c = c_i) \mid c_i \in cs\} \cup \tag{3.15}$$

$$\{(c = c_i, s_{0c}) \mid c_i \in cs, s_{0c} \in S_{0c_i}\} \cup (\bigcup_{i=1}^{n} R_c) \tag{3.16}$$

Finally, we define the labelling function by parts:

$$\begin{aligned} L(s_0) &= \{\mathit{init} = \mathit{true}\} \\ L(s) &= \{c = c_i\} \qquad \text{iff} \qquad (\mathit{init} = \mathit{true}, s) \in R \\ L(s) &= \textstyle\bigcup_{i=1}^{n} L_{c_i}(s) \qquad \text{otherwise} \end{aligned}$$

Now, we proceed to prove that the Kripke structure of Definition 3.2 stores the program that satisfies the user specification iff this program exists.

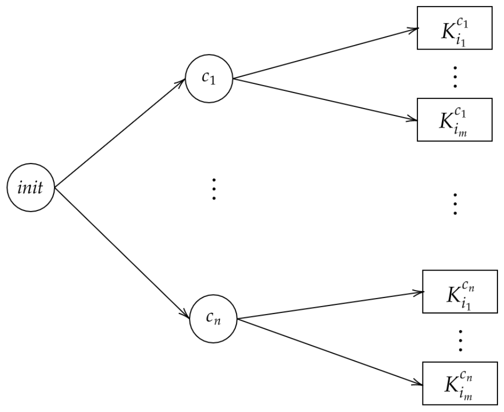

Figure 3.3: General form of the Kripke structure that encodes the filling in of a sketch.

Theorem 3.1 (reCipe correctness)**.** *Let φ be a user specification and K_{S_p} be the Kripke structure of Definition 3.2. Then the following holds: $K_S, s_0 \models EXAX\varphi$ iff there exists a candidate c such that $K_i^c, s_i \models \varphi$ for the initial state s_i, where s_i represents the initial state of a program behavior.*

Proof. By definition of the relation of satisfiability (see Definition 2.4), the following holds:

1. $K_{S_p}, s_0 \models EX\ AX\ \varphi$ iff there exists a state s_1 such that $(s_0, s_1) \in R$ and $K, s_1 \models AX\ \varphi$.

   By construction, the states accessible from s_0 are those that have a valuation of the form $c = c_i$. Therefore, we have elected a correct candidate.

2. $K_{S_p}, s_1 \models AX\ \varphi$ iff for all states s_2, if $(s_1, s_2) \in R$ then $K, s_2 \models \varphi$.

   By construction the states s_2 are the initial states of each Kripke structure K_i^c, where i is an input. Therefore, we know that each $K_i^c, s_i \models \varphi$.

In step 1, we found a candidate c and in step 2, we proved that $K_i^c, s_i \models \varphi$ for the initial state s_i. Therefore, the theorem holds. $\boxed{\lambda}$

By Theorem 3.1, the Kripke structure K_S of Definition 3.2 contains the candidate c that we are searching for. We extract the candidate c by denying the formula $EX\ AX\ \varphi$, i.e., if we try to prove $K_S, s_0 \models \neg EX\ AX\ \varphi$, we will get a counterexample where c occurs.

Remember that reCipe takes as second input a number of iterations n, where n establishes the complexity of the programs generated by reCipe. The complexity is defined by the number of the changes of each x_i variable, i.e., we repeat the lines `l_1`, ..., `l_n` n times. Therefore, we only modify the Kripke structures K_i^c and not the candidate election. Thus, Theorem 3.1 is still valid.

The way we construct K_i^c is inefficient. In the following sections, we show how *constructKripke*, see Algortihm 6, deals with the construction of K_i^c by using the *SMV* language.

3.5 Abstraction of a program

Our main objective is to abstract a sketch of a program into a Kripke structure. Carke et al. in [9] give an abstraction of sequential programs into Kripke structures. We endow Clarke's abstraction with some syntactic sugar that results in a more comfortable way to analyze the synthesis procedure.

In [9], Clarke et al. present a transformation from a WHILE program into a transition system using a GO TO program style and first-order formulas representing the elements of the Kripke structure. Their main objective is the verification of properties for sequential programs through Kripke structures and model-checking techniques. It is important to observe that we are model-checking by using finite Kripke structures, which means that some operations are limited by the domain of the variables. For example, in Code 3.3 we reduce by one the value stored in the variable t. If $t \in \{0, \dots, 10\}$, we cannot encode expressions as $(t' = t - 1)[t := 0]$. Finally, Formula 3.17 is Clarke's representation of Code 3.3.

```
while(t > 0){
    t = t-1;
}
```

Code 3.2: Decrement of t

```
l1: if(t > 0)then{goto l2;}else{goto l3;}
l2: t = t-1;goto l1;
l3: return;
```

Code 3.3: Decrement of t, GoTo style

$$\begin{aligned}\rho =_{def} &(pc = l_1 \wedge pc' = l_2 \wedge t > 0 \wedge t' = t)\vee \\ &(pc = l_1 \wedge pc' = l_3 \wedge t \leq 0 \wedge t' = t)\vee \\ &(pc = l_2 \wedge pc' = l_1 \wedge t' = t - 1)\vee \\ &(pc = l_3 \wedge pc' = l_3 \wedge t' = t)\end{aligned} \tag{3.17}$$

We now make an observation: due to the fact that the labelling function is the same as Clarke et al. give in [9], for all the Kripke structures we define, we do not mention the labelling function in the rest of the thesis.

A Kripke structure $K = (S, I, R)$ that abstracts away the behavior of Algorithm 3.3 is shown in Figure 3.4 and formally defined as follows:

$$\begin{aligned} S &= PC \times T \\ PC &= \{(pc = l) \mid l \in \{l_1, l_2, l_3\}\} \\ T &= \{(t = i) \mid i \in \{0, \ldots, 10\}\} \\ I &= \{(pc = l_1, t = 10)\} \\ R &= \{(s, s') \mid (s(v), s'(v)) \models \rho, v \in S, v' \in S'\} \end{aligned} \tag{3.18}$$

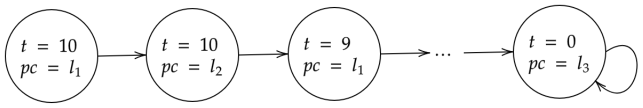

Figure 3.4: Diagram of the Kripke structure defined by Formulas 3.18 and 3.17.

The details of how we can verify $K \models \varphi$ for any CTL formula φ are available in [9, 15]. In the next section, we introduce syntactic sugar in order to define our Kripke structures in a SMV language style [17].

3.5.1 Syntactic sugar for the transitions

Formulas similar to Formula 3.17 represent the transitions over the states of a particular system [9]. However, from a programming point of view, it is easier to follow the behavior of a program line-by-line, such as Code 3.3.

McMillan in [17] defines the *SMV* (Symbolic Model Verification) language. The SMV language provides expressions for a concise representation of a Kripke structure and other entities in model checking. In our particular case, we are interested in the *case* expressions. The behavior of the *case* given by McMillan is similar as the *case* expressions of some programming languages such as Haskell or Java.

The *case* expression has the following syntax:

$$\begin{array}{ll} case & \\ & guard_1 : op_1; \\ & guard_2 : op_2; \\ & \qquad\vdots \\ & guard_n : op_n; \\ & \quad 1 : esc; \\ esac; & \end{array}$$

where for every i, op_i is the result of a $guard_i$ condition. In case of no $guard_i$ is true (denoted as 1), the result is esc.

Having the possibility to make choices by a condition, we use the assign operator := indicating the value of a particular element. For example, the assign style of Formula 3.17 is the following:

$$\begin{array}{rcl} pc' & := & case \\ & & pc = l_1 \wedge t > 0 : l_2; \\ & & pc = l_1 \wedge t \leq 0 : l_3; \\ & & pc = l_2 : l_1; \\ & & pc = l_3 : l_3; \\ & & 1 : pc; \\ & & esac; \\ & & \\ t' & := & case \\ & & pc = l_2 : t - 1; \\ & & 1 : t; \\ & & esac; \end{array}$$

Whereas the pc' case is showing the behavior of the program line-by-line, the t' case is defining the behavior of t variable. For example, the value stored in t only changes when $pc = l_2$.

The SMV style gives us the following advantages:

1. We can observe the behavior of a specific element instead of reading and analyzing a formula. In other words, this representation is mind-cheaper.
2. From a programming point of view, it is more natural to use the case expression instead of a logic formula.

Having an idea of how the assign-style works, let us formally define the syntax by unfolding the assign operator and *case* expressions into logic formulas.

Definition 3.3 (Assign style)**.** Let x be a variable, $g_1, \dots, g_n$ be boolean conditions, and $op_1, \dots, op_n, esac$ be values of x. We define the assign-style as:

$$\left.\begin{array}{rcl} x & := & case \\ & & g_1 : op_1; \\ & & \vdots \\ & & g_n : op_n; \\ & & 1 : esc; \\ & & esac; \end{array}\right\} =_{def} (g_1 \wedge x = op_1) \vee \cdots \vee (g_n \wedge x = op_n) \vee (\bigwedge_{i=1}^{n} \neg g_i \wedge x = otherwise)$$

In case of having more than one assign, we only make the conjunction of the formulas obtained from the previous definition.

Definition 3.4 (Assigns)**.** Let $x_1, \ldots, x_n$ be variables and $c_1, \ldots, c_n$ be the case analysis of each x_i. We define a multiple assign as:

$$\left.\begin{array}{ccc} x_1 & := & c_1; \\ & \vdots & \\ x_n & := & c_n; \end{array}\right\} =_{def} \bigwedge_{i=1}^{n} assign(x_i, c_i)$$

where $assign(x, c)$ is the unfolded assign as we establish in Definition 3.3.

The next result allows us to use this new style instead of the first-order formulas defined by Clarke et al. in [9].

Proposition 3.1. *Let A be a multiple assign-style such as Definition 3.4. There exists a formula transition φ such that $A \equiv \varphi$.*

Proof. The result is direct by induction on the number of variables and assigns. In each step of the induction, we unfold Definition 3.3 and Definition 3.4. And finally, we compute the disjunctive normal form. $\boxed{\lambda}$

With the new notation, we can define the elements of a Kripke structure in an assign-style.

The last definition allows us define a Kripke structure in a concise way. For example, we redefine the Kripke structure of Equation 3.18 as the Equation 3.19.

$$\begin{aligned} S &= PC \times T \\ PC &= \{(pc = l) \mid l \in \{l_1, l_2, l_3\}\} \\ T &= \{(t = i) \mid i \in \{0, \ldots, 10\}\} \\ R &= \{(s, s') \mid \forall v \in \{pc, t\}.\forall v' \in \{pc', t'\}.(s(v), s'(v')) \models C\} \\ C &= \left\{\begin{array}{ccl} pc & := & l_1; \\ t & := & 10; \\ pc' & := & case \\ & & pc = l_1 \wedge t > 0 : l_2; \\ & & pc = l_1 \wedge t \leq 0 : l_3; \\ & & pc = l_2 : l_1; \\ & & pc = l_3 : l_3; \\ & & 1 : pc; \\ & & esac; \\ t' & := & case \\ & & pc = l_2 : t - 1; \\ & & 1 : t; \\ & & esac; \end{array}\right. \end{aligned} \tag{3.19}$$

In the next section, we define Kripke structures that represents the synthesis procedure.

3.5.2 A sketch inside a Kripke structure

As we mentioned, Solar-Lezama in [23] defines a sketch as an incomplete program with assertions. For example, Code 3.4 is a sketch of the program that decrements the value stored in t.

```
while(t >= 0){
    t = ??;
}
assert(t == 0);
```

Code 3.4: A Sketch of the function that decrements the variable t.

Using the same procedure as the illustrative example, we define a Kripke structure KS that represents Code 3.4. Formula 3.20 is the transition formula of KS and Code 3.5 is the implementation of Formula 3.20.

$$\rho_S =_{def} \begin{cases} \begin{array}{lll} pc' & := & case \\ & & pc = l_1 \wedge t > 0 : l_2; \\ & & pc = l_1 \wedge t \leq 0 : l_3; \\ & & pc = l_2 : l_1; \\ & & pc = l_3 : l_3; \\ & & 1 : pc; \\ & & esac; \\ \\ t' & := & case \\ & & pc = l_2 : ??; \\ & & 1 : t; \\ & & esac; \end{array} \end{cases} \tag{3.20}$$

```
    l1:if(t > 0){goto l2;}else{goto l3;}
    l2:t = \mathtt{??};goto l1;
    l3:return;
```

Code 3.5: A Sketch of decrement in a `GoTo` style

In case we want to verify some property, there are two problems with ρ_S. The first problem is that there is not interpretation for the hole (`??`). The second one is that we want to verify a user specification against the behavior of the program with every input of the domain. Similarly as Morgenstern and Schneider do in [19], we add new states that have expressions for filling in the hole. This last strategy, provides us with the inputs to analyze different behaviors of a program.

Let us assume that the gray states in Figure 3.2 represent the particular Kripke structure K_{10} whose transition relation is defined by $\rho_S[?? \mapsto t-1][t \mapsto 10]$. If we replace $\rho_S[?? \mapsto t-1][t \mapsto 10]$ by $\rho_S[?? \mapsto t-1][t \mapsto 9]$, we obtain a different Kripke structure K_9. In Figure 3.5 we show different Kripke structures $K_0, K_1, \ldots, K_9, K_{10}$ where the transition relation of each K_i is defined by $\rho_S[?? \mapsto t-1][t \mapsto i]$. In other words, we have a different Kripke structure for each possible input.

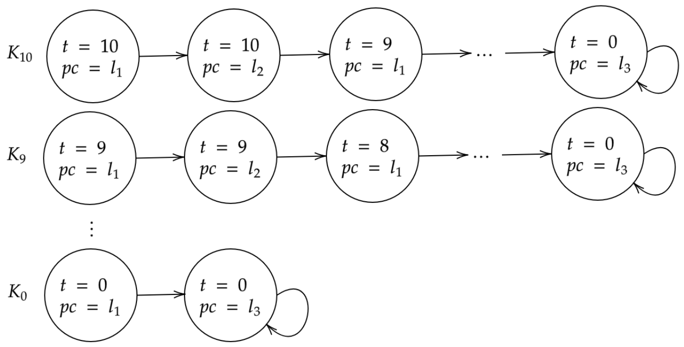

Figure 3.5: Different values of t.

Analogously to the previous case, if we calculate $\rho_S[?? \mapsto t+1][t \mapsto i]$ where $i \in \{0, \ldots, 10\}$, we generate a variety of Kripke structures $K_{e_1}, \ldots K_{e_{10}}$. For example, in Figure 3.6 in the blue box are the Kripke structures whose transition formula is defined by the substitution $[?? \mapsto t-1]$ and in the green box are the Kripke structures whose transition formula is defined by the substitution $[?? \mapsto t+1]$. Using this strategy we can build a variety of Kripke structures in the synthesis step q_1. We only need to define a set of possible set of expressions $\mathcal{E}$ and use the elements of $\mathcal{E}$ for the substitution of the hole.

We have a set of integers $\mathcal{D}$, a set of variables $\mathcal{V}$ and, the maximum height of the abstract syntax trees. We use the previous elements for the creation of arithmetic expressions using a bottom-up strategy. The grammar that we use in this example is defined in Equation 3.21. However, it can be extended for more complicated expressions.

$$e ::= n \mid v \mid e+e \mid e \times e \mid e-e \tag{3.21}$$

Where $n \in \mathcal{D}$ and $v \in \mathcal{V}$.

Once we have computed $\mathcal{E}$, we need to simulate the selection of the expression that replaces the hole and the behavior of a program by using all the inputs and the selected expression. If we suppose that q_1 has all the possible behaviors for each input, a previous step of the synthesis, called q_{sel}, would be the selection of a specific expression. We create states of the form $(?? = e)$ where

$e \in \mathcal{E}$ and a transition from q_{sel} to q_{sel_i}. We give an example of this new synthesis state in Figure 3.7.

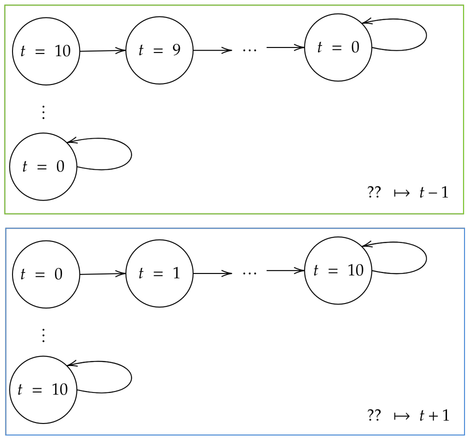

Figure 3.6: Different values for the hole create different programs.

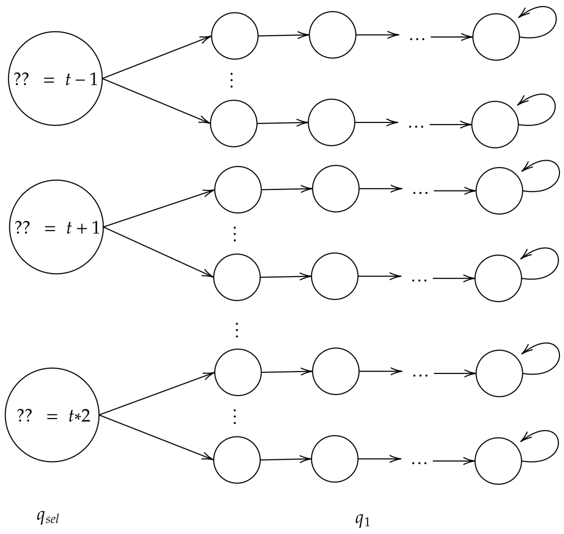

Figure 3.7: Example of the synthesis state q_{sel}.

Finally, the synthesis state q_0 is the first step of the synthesis, where all values of the variables are unknown. We denote with a star ($\star$) multiple states; also we create a transition from q_0 to all the states that appear in q_{sel}. In Figure 3.8, we give an idea of the Kripke structure that represents a solution for the sketch in Code 3.5. Observe that q_0 has a new variable *location*; this variable is the formalization of how we represent the synthesis states inside the Kripke structure.

Once we give a view of how we define the Kripke structures for the synthesis, at the end of this section, we formally define these Kripke structures.

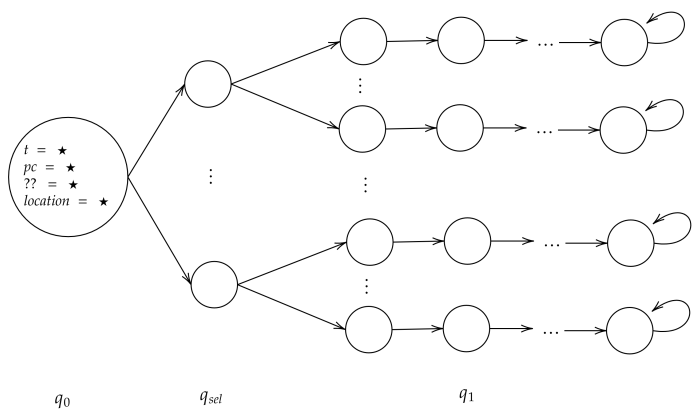

Figure 3.8: Example of the synthesis state q_0.

3.6 Kripke structure with oracles

As Morgenstern and Schneider do in [19], we call the previously Kripke structures *Kripke structure with oracles*. Our main difference is creating more specialized oracles that decide the determinism of a `goto`-statement.

```
l1:if(g){goto l2;}else{goto l3;} //g is a guard given by the user
l2:x = ??;goto l1;
l3:return;
```

Sketch 3.4: A simple sketch

One of the simplest programs that reCipe can synthesize is Sketch 3.4. This last example will be useful in order to define *Kripke structure with oracles* for more complicated programs. We present these last Kripke structures in the next section. In Definition 3.5 we formally define a Kripke structure with oracles for sketches structured as Sketch 3.4.

Definition 3.5 (Kripke structure with oracles)**.** Let g be a conditional guard, $\mathcal{V}$ and $\mathcal{E}$ be the set of variables and expressions constructed by the user specification, respectively. A *Kripke structure* *with oracles* is a Kripke structure $K = (S, S_0, R)$ where:

$$\begin{aligned}
S &= PC \times Vars \times Exps \times Location \\
PC &= \{(pc = l) \mid l \in \{l_1, \dots, l_n\}\} \\
Vars &= \{(v = i) \mid v \in \mathcal{V}, i \in \mathcal{D}\} \\
Exps &= \{(?? = e) \mid e \in \mathcal{E}\} \\
Location &= \{location = q \mid q \in \{q_0, q_{sel}, q_1\}\} \\
R &= \{(s, s') \mid \forall v \in \mathcal{V}. \forall v' \in \mathcal{V}'.(s(v), s'(v')) \models C\} \\
C &= \text{is defined by the equations: Equation 3.22 initial state, 3.23 synthesis states,} \\
&\quad \text{3.24 search space, 3.25 behavior of the program, 3.26 behavior of the variable.}
\end{aligned}$$

In Equation 3.22 there are some useful notations:

- List comprehension: The form $b : \{o_1, \dots, o_n\}$ can be unfolded as:

$$\begin{gathered}
b : o_1; \\
\vdots \\
b : o_n;
\end{gathered}$$

- $e_i(v)$: The expression $e_i(v)$ is the application of the expression e_i over the current value of v, for every $e_i \in \mathcal{E}$.

$$location \;:=\; q_0 \tag{3.22}$$

$$\begin{aligned}
location' \;:=\;& case \\
& location = q_0 : q_{sel}; \\
& location = q_{sel} : q_1; \\
& 1 : location; \\
& esac;
\end{aligned} \tag{3.23}$$

$$\begin{aligned}
??' \;:=\;& case \\
& location = q_0 : \{e_i \mid e_i \in \mathcal{E}\}; \\
& 1 : ??; \\
& esac;
\end{aligned} \tag{3.24}$$

$$
\begin{aligned}
pc' \quad := \quad & case \\
& location = q_1 \wedge pc = l_1 \wedge g : l_2; \\
& location = q_1 \wedge pc = l_1 \wedge \neg g : l_3; \\
& location = q_1 \wedge pc = l_2 : l_1; \\
& location = q_1 \wedge pc = l_3 : l_3; \\
& 1 : pc; \\
& esac;
\end{aligned}
\tag{3.25}
$$

$$
\begin{aligned}
v' \quad := \quad & case \\
& location = q_{sel} : \{i \mid i \in \mathcal{D}\}; \\
& location = q_1 \wedge ?? = e_1 \wedge pc = l_2 : e_1(v); \\
& \vdots \\
& location = q_1 \wedge ?? = e_n \wedge pc = l_2 : e_n(v); \\
& 1 : v; \\
& esac;
\end{aligned}
\tag{3.26}
$$

Observe that the oracle variables are those that occur in the right-hand side of the valuation $location = q_0$. Therefore, the set `Exps` is the set of oracle variables.

Let us show an example of how to construct a Kripke structure with oracles by analyzing a user specification.

Example 3.1. *The user wishes a program that decrements by one the value of a variable t. The formal specification is the following:*

$$\forall x.(EF\ (t = 0) \wedge \forall.x(t > x \wedge x > 1 \rightarrow EF\ (t < x \wedge t > 0))) \tag{3.27}$$

The guard of the program is $t > 0$ and the number of iterations for the buttom-up algorithm is n. The set of variables and integers domain of the program are:

$$\mathcal{V} = \{t\} \tag{3.28}$$

$$\mathcal{D} = \{0, \ldots, 5\} \tag{3.29}$$

According to the Equations 3.28, 3.29 and the number of iterations, the set of possible expressions *is:*

$$\begin{aligned}
&\mathcal{E}_0 = \{0, \ldots, 5, t\}\\
&\mathcal{E}_1 = \mathcal{E}_0 \cup \{t_1 \star t_2 \mid t_1, t_2 \in \mathcal{E}_0, \star \in \{+, \times, -\}\}\\
&\vdots\\
&\mathcal{E} = \mathcal{E}_n = \mathcal{E}_{n-1} \cup \{t_1 \star t_2 \mid t_1, t_2 \in \mathcal{E}_{n-1}, \star \in \{+, \times, -\}\}
\end{aligned} \tag{3.30}$$

Using the previous information, we construct the Kripke structure sketch 3.31.

$$\begin{aligned}
S &= PC \times Vars \times Exps \times Location\\
PC &= \{(pc = l) \mid l \in \{l_1, l_2, l_3, l_4\}\} \cup \{pc = \star\}\\
Vars &= \{(v = i) \mid v \in \mathcal{V}, i \in \mathcal{D}\} \cup \{v = \star\}\\
Exps &= \{(?? = e) \mid e \in \mathcal{E}\} \cup \{?? = \star\}\\
Location &= \{location = q \mid q \in \{q_0, q_{sel}, q_{sel_i}, q_1\}\}\\
C &= \textit{as the Definition } 3.5 \textit{ establishes by replacing}\\
&\quad \mathcal{V}, \mathcal{D} \textit{ and } \mathcal{E} \textit{ by Equations } 3.28, 3.29 \textit{ and } 3.30\textit{, respectively.}
\end{aligned} \tag{3.31}$$

Now, let us generalize the Kripke structure with oracles by filling in Sketch 3.3.

3.6.1 Generalizing the Kripke structures with oracles

As we said in Section 3.4, Sketch 3.3 has *different* types of metavariables: expression holes, guards holes, and non-deterministic *goto*-statements. These metavariables can be replaced by concrete expressions, predicates and deterministic `goto`-statements, respectively. Furthermore, we can consider the expressions and guards holes as non-deterministic variables. We show an example of this non-determinism in Equation 3.24.

In order to generalize the Kripke structures with oracles, see Definition 3.5, we define the following sets:

- For every expression hole `e_`i, we create a set $Exps_i$, where $Exps_i$ is a set of concrete expressions that can replace `e_`i.
- For every guard hole `g_`i, we create the set $\mathcal{G}$ that is a set of predicates that can replace `g_`i.
- For every non-deterministic `goto`-statement, we create the set $\mathcal{GT}$ that is a set of line labels that can replace a non-deterministic `goto`-statement.

We can extend the current definition of the Kripke structures with oracles, if we *add* the following to Definition 3.5:

- We redefine the set $Exps$ of S, as:

$$Exps = Exps_1 \times \cdots \times Exps_n \times Guards_1 \times \cdots \times Guards_n \times GT_1 \times \cdots \times GT_n$$

- We create an assign, see Definition 3.3, for each metavariable such as in Equation 3.24.

- In Equation 3.25, we need to create every possible jump. For example, if we are in line 1 and g_1 can be replaced by a predicate p but can also be replaced by a predicate q, then we need to consider the following cases:

$$\begin{gathered} location = q_1 \wedge pc = l_1 \wedge g_1 = p \wedge p : l_2; \\ location = q_1 \wedge pc = l_1 \wedge g_1 = p \wedge \neg p \wedge dgt_1 = l_2 : l_2; \\ location = q_1 \wedge pc = l_1 \wedge g_1 = p \wedge \neg p \wedge dgt_1 = l_1 : l_1; \\ location = q_1 \wedge pc = l_1 \wedge g_1 = q \wedge p : l_2; \\ location = q_1 \wedge pc = l_1 \wedge g_1 = q \wedge \neg q \wedge dgt_1 = l_2 : l_2; \\ location = q_1 \wedge pc = l_1 \wedge g_1 = q \wedge \neg q \wedge dgt_1 = l_1 : l_1; \end{gathered} \tag{3.32}$$

  where dgt_1 is the variable that represents an unique jump, i.e., a determinisitic `goto`-statement.

- This case is analogous to the previous one. We need to consider every possible expression hole `e_i`.

In Appendix A, we show an example of a Kripke structure with oracles for two variables and one iteration. In the previously mentioned code we show we replace the synthesis of guards by angelic determinism. In other words, we assume the existence of the guards. This last approach is an optimization that we describe in the next chapter.

It is clear that this construction will cause the state-explosion problem to occur. In the next section, we present the complexity of reCipe in terms of the size of the search space and verify this last assertion.

3.7 Complexity of reCipe

An important aspect of algorithms is their complexity. Whereas the complexity of model-checking algorithms depends on the size of the model, the complexity of program-synthesis depends on the size of the space of search. First, let us present the complexity of CTL model checking (see Theorem 3.2).

Theorem 3.2 (Complexity of CTL model checking [9]). *Let φ be a CTL formula and $K = (S, S_0, R, L)$ be a Kripke structure. There is an algorithm for determining whether φ is true in a state s of K that runs in time:*

$$O(|\varphi| \cdot (|S| + |R|)) \tag{3.33}$$

where $|\varphi|$ is the size of the formula φ.

In our particular case, the Kripke structures with which we are working have a generic form; see Figure 3.2. We can give a better bound using the generic form. First, we construct the bounds in a intuitive way and then we formalize the complexity.

By construction, the set of states is divided in four sections (*synthesis states*):

1. q_0: A unique state.
2. q_{sel}: The number of states is equal to the number of expressions generated by the bottom-up search.
3. q_1: The number of states is equal to the number of states generated by the unfolding of the program behavior for each possible replacement of the holes of the program.

Using the partition generated by the synthesis states, we define the number of states as follows:

Definition 3.6. Let $\mathcal{R}$ be the set of replacements of the metavariables generated by a user specification. The number of states of a Kripke structure sketch is:

$$|S| = 1 + |\mathcal{R}| + B_P \tag{3.34}$$

where B_P is the number of states of each program behavior.

In order to give an exact number for $\mathcal{R}$, we need to compute the number of possible concrete expressions, predicates and to know the number of deterministic `goto`-statements.

We denote the set of concrete expressions as E. We define E by using a bottom-up strategy i.e., we create all possible expressions according to a bounded abstract syntax tree. Using this information, we can compute $|\mathcal{E}|$ as follows:

Definition 3.7. Let $\mathcal{V}$ be the set of variables, $\mathcal{D}$ be the set of inputs, both generated by a user specification and, n be the number of iterations. The cardinality of $\mathcal{E}$ is:

$$\begin{aligned} |\mathcal{E}| &= |\mathcal{E}_n| \text{where:} \\ |\mathcal{E}_0| &= |\mathcal{V}| + |\mathcal{D}| \\ |\mathcal{E}_1| &= |\mathcal{E}_0| + \binom{|\mathcal{E}_0|}{2} \\ &\vdots \\ |\mathcal{E}_n| &= |\mathcal{E}_{n-1}| + \binom{|E_{n-1}|}{2} \end{aligned} \tag{3.35}$$

The number of guards is analogous to $|\mathcal{E}|$. Let us denote the set of predicates as G. In the worst case, we can assume that all the expressions have the same type. This will cause G to have a similar cardinality of E but instead of using as base the sets of variables and inputs, we use E.

Finally, we need to compute the number of labeled lines that represent a deterministic `goto`-statement. Let us denote the number of the labeled lines as L. Cause we can have two possibilities for each labeled line, the number of possible lines is the number of variables times the number of iterations n:

$$|L| = |\mathcal{V}| \times n \times 2$$

We can observe that the number of states explodes. Therefore, if we want to reduce the number of states, we need to define some strategies so that we can improve the model checking process. We analyze these last problems in the next chapter of the thesis.

Chapter 4

Experiments and optimizations

"(...) it is not clear than we can always select a 'best' algorithm (...) Perhaps it is time to abandon the 'winner-takes-all' mentality that seems to pervade algorithmic research."

Vardi, Moshe Y. (2007) [25]

The execution of reCipe basically relies on the labelling algorithm (see Algorithm 1). We can ensure that reCipe works for complex problems, if we are able to ensure that the labelling algorithm works for this kind of problems in spite of the *state-explosion* problem (see [9, 15]). We list some experiments and projects that use variations of the labelling algorithm and have given good results:

- Burch et al. in [4] tested a μ-calculus labelling algorithm for verifying CTL properties in a model with 10^{20} states. This is a classical example to show that the *model checking* is viable for complex problems.
- The model checker Kratos [8] can verify properties of sequential C-programs. Kratos is built on top of nuXmv.

We show some executions of reCipe and show that we need optimizations over the naive approach of the previous chapter. If we want to optimize the executions of reCipe, we need to create strategies that reduce the size of the model created by reCipe and reduce the number of nodes that occur in the OBDD created by the labelling algorithm.

In order to show our optimizations, we present an execution of reCipe in a toy example. Besides the fact that we use a toy example, the specification forces to search in the majority of states generated by reCipe. Then, we can infer some optimizations.

For our empirical experiments, we want to synthesize the program that satisfies the following specification:

$$\bigwedge_{x \in V} (EF\ x = 0\ \wedge \forall i.(x > i \wedge x > 1 \rightarrow (EF\ x < i \wedge x > 0))) \tag{4.1}$$

where V is a set of variables and $i \in \{0, \dots, 4\}$. Equation 4.1 establishes the behavior of a program P that decrements the variables that occur in V. In other words, if $V = \{x_1, x_2, x_3\}$, then P decrements the value of x_1. After that, P decrements the value of x_2. Finally, P decrements the value of x_3. For our experiments, we assume the following:

1. $|V|$ is 1, 2, 3, 4, 5 and 8.
2. We construct candidate expressions as in Appendix A, i.e., we only construct addition and subtraction of the form $x + n$, $x - n$, $n - x$ where x is a variable and $n \in \{0, \dots, 4\}$.
3. We construct three candidate guards $x < 0$, $x > 0$ and $x = 0$, where x is a variable.

The *jump* we make between five variables and eight variables is for showing how the number of states blows up in spite of using our optimizations.

4.1 A first evaluation

In our first approach, we synthesize all the holes using a same `nuXmv` script. In other words, we try to synthesize all the missing values in order to complete our imaginary sketch (see Chapter 3). In Figure 4.1, red graph, and Table 4.1, we observe how easily the number of states blows up. In fact, when we have five variables to synthesize, `nuXmv` takes more than three seconds to get the desired counterexample.

4.2 Modular Synthesis

Bodik et al. in [2] define the *angelic non-determinism* as the operator that makes it possible for a program to meet its specification. In other words, such non-determinism allows the program to end and to satisfy a specification. We can use the angelic operator (*) for splitting the synthesis procedure in two steps:

1. Synthesize the structure (deterministic `goto`-statements) and the holes expressions (`expi` where `i` is the number of lines of the sketch).
2. Synthesize the guards where there were angelic non-determinism.

This partition is straightforward; it is a similar to that idea of the non-deterministic `goto`-statements. We assume the existence of a guard and allow the program to take both branches of the `if`-conditionals. An example of this angelic non-determinism is shown in Appendix A. In Figure 4.1, blue graph, we show the time that reCipe takes to synthesize the first step of the modular synthesis.

4.3 Ordering the variables

As we mention in Chapter 2, one of the optimizations of the labelling algorithm is ordering the variables. Although finding a good ordering is NP-complete [9, 15], in our particular case we can give a good variable ordering.

On the one hand, nuXmv has some predefined heuristics for ordering variables [5]. For example, nuXmv offers an genetic algorithm called "genetic" or another algorithm that stores only one OBDD at a time, i.e., a dynamic ordering called "exact", etc. On the other hand, we know that our Kripke structures with oracles have a search-tree-like form. We can take advantage of that form by giving the following order:

$$location < cycle_1 < \cdots < cycle_n < pc < exp_0 < x_0 < exp_1 < x_1 < \cdots < exp_n < x_n \tag{4.2}$$

where each $cycle_i$ is a variable that represents the value of a deterministic goto-statement, pc is the program counter, each exp_i, x_i is the concrete expression that changes the value of the variable x_i.

In Figure 4.1, the black graph corresponds to the ordering *exact* provided by nuXmv and the magenta graph corresponds to our ordering. The advantage of our ordering over the nuXmv is evident. Another advantage of our ordering is that it can be applied to any number of boolean variables.

4.4 The results

In this section, we show the results of the executions of reCipe in our toy example in two forms: a graph and a table. All the results we show were run on a machine with a i-5 10th generation Intel processor (4 cores of 1 GHz processor base frequency) and 4 GB RAM.

We observe that our optimizations give obvious advantages over the naive approach. In fact, synthesizing the guards will have a similar behavior. In the next chapter, we give some ideas for heuristics that could give better results.

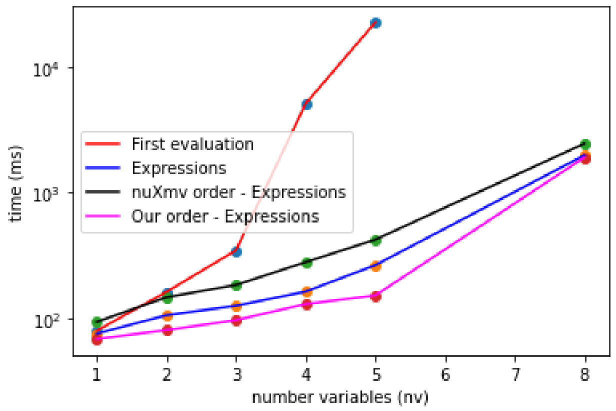

Figure 4.1: Behavior of reCipe with different optimizations. The red graph corresponds to the naive approach. The blue graph is partitioning the variables into angelic and non-angelic variables. The black graph corresponds to the nuXmv "exact" ordering. The magenta graph corresponds to our ordering.

Optimization	1 variable		3 variables	
	OBDD nodes	Synthesis time	OBDD nodes	Synthesis time
Naive	11977	∼79 ms	371818	∼345 ms
Partitioned	8053	∼75 ms	71071	∼125 ms
nuXmv ordering	2709	∼100 ms	8743	∼183 ms
Our ordering	5817	∼68 ms	14860	∼96 ms

Optimization	5 variables		8 variables	
	OBDD nodes	Synthesis time	OBDD nodes	Synthesis time
Naive	837682	∼22427 ms	-	-
Partitioned	237686	∼263 ms	954481	∼1967 ms
nuXmv ordering	15160	∼420 ms	24906	∼2436 ms
Our ordering	78400	∼151 ms	645075	∼1887 ms

Table 4.1: reCipe statistics

Chapter 5

Conclusion and future work

"I do not need sleep. I need answers."

Sheldon C. - The Big Bang Theory (2009) [16]

reCipe is an extension of the procedure given by Morgenstern and Schneider in [19]. This extension takes advantage of the semantics of CTL and the non-determinism offered by the Kripke structures with oracles. Our evaluation showed that we can synthesize small programs from a temporal formula. Moreover, we can make modular synthesis by replacing the guards by angelic non-deterministic operators. Also, we fixed the correctness of the procedure originally given by Morgenstern and Schneider and formalized the Kripke structures with oracles by adding syntactic sugar to the formulas that represents a Kripke structure.

There are some possible future directions for the reCipe approach:

1. This thesis only synthesizes sequential programs. We believe that reCipe is capable of synthesizing concurrent programs by adding new program counters for each thread, and general properties of concurrent programs. We can add this last properties as fairness constraints.
2. If we are able to create a set of temporal constraints from the user specification and add these constraints to the Kripke structure oracle creation, we will make the synthesis more efficient.
3. This thesis relies on the procedure of a complete an imaginary generic sketch[1]. Synthesizing transition machines as `Strix` does in [18], we could generate a sketch with the behavior established by the user specification.
4. The labelling algorithm analyzes a CTL formula by using a fix-point characterization. This characterization allows us to *see* a temporal formula as a sequence of conjunctions and disyunctions. We think that we can add the process of the labelling algorithm (LA) to CEGIS [23]. This extension could allow us to create sketches with `assert`-statements that can have temporal operators. The interesting question of this idea is how to represent the EX and AX operators, and whether or not CEGIS+LA converges in a reasonable time.

[1]Imaginary in the sense that the generic sketch does not appear in the procedure.

Appendix A

An example of synthesis via **nuXmv**

Let us assume that the user gives the following specification:

$$\varphi = \varphi_1 \land \varphi_2 \tag{A.1}$$

$$\varphi_1 = (EF\ t = 0\ \land \forall i.(t > i \land i > 1 \rightarrow (EF\ t < i \land t > 0))) \tag{A.2}$$

$$\varphi_2 = (EF\ r = 0\ \land \forall i.(r > i \land i > 1 \rightarrow (EF\ r < i \land r > 0))) \tag{A.3}$$

Following the idea of Algorithm 5, we compute the following sets:

$$\mathcal{V} = \{t, r\}$$
$$\mathcal{D} = \{0, \ldots, 5\}$$

Let us keep the idea that we need to filling in a generic sketch. In Code A.2, we encode the process of filling in Sketch A.1.

```
void f(int* x, int* y){
l1: if(*) then {goto l2;} else {goto l3;}
l2: t = exp0; goto {l1,l3};//cycle1
l3: if(*) then {goto l4;} else {goto l5;}
l4: t = exp1; goto {l3,l5};//cycle2
l5: return;
}
```

Sketch A.1: A sketch that we need to fill in.

The `if`-conditionals of lines `l_1` and `l_3` have a star (*). This star allows an angelic non-determinism over our synthesis procedure. As a consequence, we can split the synthesis procedure in two steps: first, we synthesize the concrete expressions and deterministic `goto`-statements. Second, we synthesize the guards that replace the stars. For further details, see Chapter 4.

```
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    location = q0
    pc1 = 1
    t = 0
    cycle1 = 1
    exp0 = vt
    r = 0
    cycle2 = 3
    exp1 = vr
  -> State: 1.2 <-
    location = qsel
    exp0 = mvtn1
    exp1 = mvrn1
```

Response A.1: First step of the synthesis.

Let us explain some of the notations we use in Code A.2:

- The variables `cycle1` and `cycle2` represent the line that replaces the non-deterministic `goto`-statement.
- The variable `ni`, where `i` $\in \{0, \ldots, 5\}$, represents the natural number `i`.
- The variables `vt` and `vr` represent the variables t and r, respectively.
- The variable `plxy`, where `x` and `y` are numbers or variables, represents the concrete expression $x + y$.
- The variable `mxy`, where `x` and `y` are numbers or variables, represents the concrete expression $x - y$.
- We do not show all the values in `exp0` and `exp1`. The missing expressions are combinations of the `ni`'s, `vt` and `vr` for the `pl` and `vr` prefixes.

If we input Code A.2 into nuXmv, we obtain Response A.1. Observe that the values of variables cycle1, cycle2 of State 1.1, and the values of variables exp0, exp1 of State 1.2 are the values that complete Sketch A.1. Now, we have to complete Sketch A.2.

```
void f(int* x, int* y){
l1: if(guard0) then {goto l2;} else {goto l3;}
l2: t = t - 1; goto l1;
l3: if(guard1) then {goto l4;} else {goto l5;}
l4: t = r - 1; goto l3;
l5: return;
}
```

Sketch A.2: A sketch where we need synthesize predicates.

To get a complete program, we fill in Sketch A.2 replacing the holes guard0 and guard1 by the values we get from the model checker.

In Code A.2, we encode each variable behavior under a arithmetic expression. In Code A.3 we do a similar thing, we encode all the program behavior under a guard election. The reason of the last reasoning is that the guards can affect all the program behavior.

As the last call of nuXmv, we input Code A.3 into the model checker and get Response A.2.

As last response, we replace the holes guard0 and guard1 by the last occurrence of the variables in Response A.2. By this last, we can generate the desired program (see Code A.3).

```
void f(int* x, int* y){
l1: if(t > 0) then {goto l2;} else {goto l3;}
l2: t = t - 1; goto l1;
l3: if(r > 0) then {goto l4;} else {goto l5;}
l4: t = r - 1; goto l3;
l5: return;
}
```

Code A.1: The desired program.

```
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    location = q0
    pc1 = 1
    t = 0
    r = 0
    guard0 = gvtn0
    guard1 = gvrn0
  -> State: 1.2 <-
    location = qsel
```

Response A.2: First step of the synthesis.

```
MODULE main
VAR
        location:{q0,qsel,q1};
        pc1:{1,2,3,4,5};

        t:{0,1,2,3,4,5};
        cycle1:{1,3};
        exp0:{vt,n0,n1,n2,n3,n4,n5,
              plvtvt,plvtn0,plvtn1,plvtn2,plvtn3,...,
              mvtvt,mvtn0,mvtn1,mvtn2,mvtn3,...,mn5n5};

        r:{0,1,2,3,4,5};
        cycle2:{3,5};
        exp1:{vr,n0,n1,n2,n3,n4,n5,
              plvrvr,plvrn0,plvrn1,plvrn2,plvrn3,...,
               mvrvr,mvrn0,mvrn1,mvrn2,mvrn3,...,mn5n5};

INIT
        location = q0;

ASSIGN
        next(location) := case
                location = q0 : qsel;
                location = qsel : q1;
                location = q1 : q1;
                TRUE : location;
        esac;

        next(cycle1) := case
                location = q0 : {1,3};
                TRUE : cycle1;
        esac;

        next(cycle2) := case
                location = q0 : {3,5};
                TRUE : cycle2;
        esac;

        next(pc1) := case
                location = qsel : 1;
                location = q1 & pc1 = 1 : {2,3};
                location = q1 & pc1 = 2 : cycle1;
                location = q1 & pc1 = 3 : {4,5};
                location = q1 & pc1 = 4 : cycle2;
                location = q1 & pc1 = 5 : 5;
                TRUE : pc1;
        esac;


        next(exp0) := case
                location = q0 :
                    {vt,n0,n1,n2,n3,n4,n5,
                     plvtvt,plvtn0,plvtn1,plvtn2,plvtn3,plvtn4,plvtn5,
                     pln0vt,pln0n0,pln0n1,pln0n2,pln0n3,pln0n4,pln0n5,
                         --The rest of the cases of the ASSIGN ... };
                TRUE : exp0;
        esac;


        next(exp1) := case
                location = q0 :
                    {vr,n0,n1,n2,n3,n4,n5,
                     plvrvr,plvrn0,plvrn1,plvrn2,plvrn3,plvrn4,plvrn5,
                     pln0vr,pln0n0,pln0n1,pln0n2,pln0n3,pln0n4,pln0n5,
                           --The rest of the cases of the ASSIGN ... };
                TRUE : exp1;
        esac;


        next(t) := case
        location = qsel : {0,1,2,3,4,5};
        location = q1 & exp0 = vt & t >= 0 & t <= 5 & pc1 = 2 : t;
        location = q1 & exp0 = n0 & 0 >= 0 & 0 <= 5 & pc1 = 2 : 0;
        location = q1 & exp0 = n1 & 1 >= 0 & 1 <= 5 & pc1 = 2 : 1;
        location = q1 & exp0 = n2 & 2 >= 0 & 2 <= 5 & pc1 = 2 : 2;
        location = q1 & exp0 = n3 & 3 >= 0 & 3 <= 5 & pc1 = 2 : 3;
        location = q1 & exp0 = n4 & 4 >= 0 & 4 <= 5 & pc1 = 2 : 4;
        location = q1 & exp0 = n5 & 5 >= 0 & 5 <= 5 & pc1 = 2 : 5;
        location = q1 & exp0 = plvtvt &
            (t + t) >= 0 & (t + t) <= 5 & pc1 = 2 : (t + t);
        location = q1 & exp0 = plvtn0 &
            (t + 0) >= 0 & (t + 0) <= 5 & pc1 = 2 : (t + 0);
        location = q1 & exp0 = plvtn1 &
        (t + 1) >= 0 & (t + 1) <= 5 & pc1 = 2 : (t + 1);
        -- The rest of cases for the plus operator and all
        -- the cases for the minus operator

        TRUE : t;
        esac;

        next(r) := case
        location = qsel : {0,1,2,3,4,5};
        location = q1 & exp1 = vr & r >= 0 & r <= 5 & pc1 = 4 : r;
        location = q1 & exp1 = n0 & 0 >= 0 & 0 <= 5 & pc1 = 4 : 0;
        location = q1 & exp1 = n1 & 1 >= 0 & 1 <= 5 & pc1 = 4 : 1;
        location = q1 & exp1 = n2 & 2 >= 0 & 2 <= 5 & pc1 = 4 : 2;
        location = q1 & exp1 = n3 & 3 >= 0 & 3 <= 5 & pc1 = 4 : 3;
        location = q1 & exp1 = n4 & 4 >= 0 & 4 <= 5 & pc1 = 4 : 4;
        location = q1 & exp1 = n5 & 5 >= 0 & 5 <= 5 & pc1 = 4 : 5;
        location = q1 & exp1 = plvrvr &
            (r + r) >= 0 & (r + r) <= 5 & pc1 = 4 : (r + r);
        location = q1 & exp1 = plvrn0 &
            (r + 0) >= 0 & (r + 0) <= 5 & pc1 = 4 : (r + 0);
        location = q1 & exp1 = plvrn1 &
            (r + 1) >= 0 & (r + 1) <= 5 & pc1 = 4 : (r + 1);
        -- The rest of cases for the plus operator and all
        -- the cases for the minus operator
```

```
        TRUE : r;
        esac;

CTLSPEC
!EX AX (((EF ((t = 0))) &
    ((((t > 0) & (0 > 1)) - >  (EF (((t < 0) & (t > 0))))) &
    ((((t > 1) & (1 > 1)) - >  (EF (((t < 1) & (t > 0))))) &
    ((((t > 2) & (2 > 1)) - >  (EF (((t < 2) & (t > 0))))) &
    ((((t > 3) & (3 > 1)) - >  (EF (((t < 3) & (t > 0))))) &
    ((((t > 4) & (4 > 1)) - >  (EF (((t < 4) & (t > 0))))) &
    (((t > 5) & (5 > 1)) - >  (EF (((t < 5) & (t > 0)))))))))))) &
    ((EF ((r = 0))) &
    ((((r > 0) & (0 > 1)) - >  (EF (((r < 0) & (r > 0))))) &
    ((((r > 1) & (1 > 1)) - >  (EF (((r < 1) & (r > 0))))) &
    ((((r > 2) & (2 > 1)) - >  (EF (((r < 2) & (r > 0))))) &
    ((((r > 3) & (3 > 1)) - >  (EF (((r < 3) & (r > 0))))) &
    ((((r > 4) & (4 > 1)) - >  (EF (((r < 4) & (r > 0))))) &
    (((r > 5) & (5 > 1)) - >  (EF (((r < 5) & (r > 0))))))))))))));
```

Code A.2: SMV code for synthesizing Equations A.1-A.3.

```
MODULE main
VAR
        location:{q0,qsel,q1};
        pc1:{1,2,3,4,5};


        t:{0,1,2,3,4,5};
    r:{0,1,2,3,4,5};


    guard0:{gvtn0,lvtn0,evtn0};
    guard1:{gvrn0,lvrn0,evrn0};

INIT
        location = q0;

ASSIGN
        next(location) := case
                location = q0 : qsel;
                location = qsel : q1;
                location = q1 : q1;
                TRUE : location;
        esac;

        next(t) := case
                location = q1 &
                (t - 1) >= 0 & (t - 1) <= 5 & pc1 = 2 : (t - 1);
                TRUE : t;
        esac;

        next(r) := case
                location = q1 &
                (r - 1) >= 0 & (r - 1) <= 5 & pc1 = 4 : (r - 1);
                TRUE : r;
        esac;

        next(guard0) := case
                location = q0 : {gvtn0,lvtn0,evtn0};
                TRUE : guard0;
        esac;
        next(guard1) := case
                location = q0 : {gvrn0,lvrn0,evrn0};
                TRUE : guard1;
        esac;

        next(pc1) := case
                location = qsel : 1;
                location = q1 : case
                        guard0 = gvtn0 & guard1 = gvrn0 : case
                                pc1 = 1 & t > 0 : 2;
                                pc1 = 1 & t <= 0 : 3;
                                pc1 = 2 : 1;
                                pc1 = 3 & r > 0 : 4;
                                pc1 = 3 & r <= 0 : 5;
                                pc1 = 4 : 3;
                                pc1 = 5 : 5;
                        esac;
                        guard0 = gvtn0 & guard1 = lvrn0 : case
                                pc1 = 1 & t > 0 : 2;
                                pc1 = 1 & t <= 0 : 3;
                                pc1 = 2 : 1;
                                pc1 = 3 & r < 0 : 4;
                                pc1 = 3 & r >= 0 : 5;
                                pc1 = 4 : 3;
                                pc1 = 5 : 5;
                        esac;
                        guard0 = gvtn0 & guard1 = evrn0 : case
                                pc1 = 1 & t > 0 : 2;
                                pc1 = 1 & t <= 0 : 3;
                                pc1 = 2 : 1;
                                pc1 = 3 & r = 0 : 4;
                                pc1 = 3 & r != 0 : 5;
                                pc1 = 4 : 3;
                                pc1 = 5 : 5;
                        esac;
                        guard0 = lvtn0 & guard1 = gvrn0 : case
                                pc1 = 1 & t < 0 : 2;
                                pc1 = 1 & t >= 0 : 3;
                                pc1 = 2 : 1;
                                pc1 = 3 & r > 0 : 4;
                                pc1 = 3 & r <= 0 : 5;
                                pc1 = 4 : 3;
                                pc1 = 5 : 5;
                        esac;
                        guard0 = lvtn0 & guard1 = lvrn0 : case
                                pc1 = 1 & t < 0 : 2;
                                pc1 = 1 & t >= 0 : 3;
                                pc1 = 2 : 1;
                                pc1 = 3 & r < 0 : 4;
                                pc1 = 3 & r >= 0 : 5;
                                pc1 = 4 : 3;
                                pc1 = 5 : 5;
                        esac;
                        guard0 = lvtn0 & guard1 = evrn0 : case
                                pc1 = 1 & t < 0 : 2;
                                pc1 = 1 & t >= 0 : 3;
                                pc1 = 2 : 1;
                                pc1 = 3 & r = 0 : 4;
                                pc1 = 3 & r != 0 : 5;
                                pc1 = 4 : 3;
```

```
                pc1 = 5 : 5;
        esac;
        guard0 = evtn0 & guard1 = gvrn0 : case
                pc1 = 1 & t = 0 : 2;
                pc1 = 1 & t != 0 : 3;
                pc1 = 2 : 1;
                pc1 = 3 & r > 0 : 4;
                pc1 = 3 & r <= 0 : 5;
                pc1 = 4 : 3;
                pc1 = 5 : 5;
        esac;
        guard0 = evtn0 & guard1 = lvrn0 : case
                pc1 = 1 & t = 0 : 2;
                pc1 = 1 & t != 0 : 3;
                pc1 = 2 : 1;
                pc1 = 3 & r < 0 : 4;
                pc1 = 3 & r >= 0 : 5;
                pc1 = 4 : 3;
                pc1 = 5 : 5;
        esac;
        guard0 = evtn0 & guard1 = evrn0 : case
                pc1 = 1 & t = 0 : 2;
                pc1 = 1 & t != 0 : 3;
                pc1 = 2 : 1;
                pc1 = 3 & r = 0 : 4;
                pc1 = 3 & r != 0 : 5;
                pc1 = 4 : 3;
                pc1 = 5 : 5;
        esac;
        TRUE : pc1;
```

```
                esac;
                TRUE : pc1;
        esac;


CTLSPEC
!EX AX (((EF ((t = 0))) &
    ((((t > 0) & (0 > 1)) - >  (EF (((t < 0) & (t > 0))))) &
    ((((t > 1) & (1 > 1)) - >  (EF (((t < 1) & (t > 0))))) &
    ((((t > 2) & (2 > 1)) - >  (EF (((t < 2) & (t > 0))))) &
    ((((t > 3) & (3 > 1)) - >  (EF (((t < 3) & (t > 0))))) &
    ((((t > 4) & (4 > 1)) - >  (EF (((t < 4) & (t > 0))))) &
    (((t > 5) & (5 > 1)) - >  (EF (((t < 5) & (t > 0))))))))))) &
    ((EF ((r = 0))) &
    ((((r > 0) & (0 > 1)) - >  (EF (((r < 0) & (r > 0))))) &
    ((((r > 1) & (1 > 1)) - >  (EF (((r < 1) & (r > 0))))) &
    ((((r > 2) & (2 > 1)) - >  (EF (((r < 2) & (r > 0))))) &
    ((((r > 3) & (3 > 1)) - >  (EF (((r < 3) & (r > 0))))) &
    ((((r > 4) & (4 > 1)) - >  (EF (((r < 4) & (r > 0))))) &
    (((r > 5) & (5 > 1)) - >  (EF (((r < 5) & (r > 0))))))))))));
```

Code A.3: SMV code for synthesizing the guards.

Bibliography

[1] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking (Representation and Mind Series)*. The MIT Press. 7

[2] Bodik, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., and Rodarmor, C. (2010). Programming with Angelic Nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 339–352, New York, NY, USA. Association for Computing Machinery. 48

[3] Bollig, B. and Wegener, I. (1996). Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002. 18

[4] Burch, J. R., Clarke, E. M., Long, D. E., McMillan, K. L., and Dill, D. L. (1994). Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424. 47

[5] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., and Tonetta, S. (2014). The nuXmv symbolic model checker. In *CAV*, pages 334–342. 18, 49

[6] Cavada, R., Cimatti, A., Jochim, C. A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., and Tchaltsev, A. (2015). *NuSMV 2.6 User Manual*. FBK-irst. 18

[7] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (1999). NuSMV: A New Symbolic Model Verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, page 495–499, Berlin, Heidelberg. Springer-Verlag. 10

[8] Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., and Roveri, M. (2011). Kratos – A Software Model Checker for SystemC. In Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification*, pages 310–316, Berlin, Heidelberg. Springer Berlin Heidelberg. 47

[9] Clarke, E. M., Grumberg, O., and Peled, D. A. (2000). *Model Checking*. MIT Press, Cambridge, MA, USA. 7, 9, 14, 15, 16, 17, 18, 21, 23, 29, 32, 33, 35, 43, 47, 48

[10] Déharbe, D. (2003). *A Tutorial Introduction to Symbolic Model Checking*, pages 215–237. 15, 16, 17

[11] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA. 21

[12] Gotzhein, R. (1992). Temporal logic and applications—a tutorial. *Computer Networks and ISDN Systems*, 24(3):203 – 218. 8

[13] Grimm, T., Lettnin, D., and Hübner, M. (2018). A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip. *Electronics*, 7:81. 7

[14] Halpern, J. Y., Harper, R., Immerman, N., Kolaitis, P. G., Vardi, M. Y., and Vianu, V. (2001). On the Unusual Effectiveness of Logic in Computer Science. *Bulletin of Symbolic Logic*, 7(2):213–236. 7

[15] Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, USA. 8, 10, 11, 12, 14, 15, 16, 17, 33, 47, 48

[16] Lorre, C. and Prady, B. (2009). The Einstein Approximation. *The Big Bang Theory*, 3(14). 51

[17] McMillan, K. L. (1993). *Symbolic Model Checking.* Kluwer Academic Publishers, USA. 33

[18] Meyer, P. J., Sickert, S., and Luttenberger, M. (2018). Strix: Explicit Reactive Synthesis Strikes Back! In Chockler, H. and Weissenbacher, G., editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 578–586. Springer. 51

[19] Morgenstern, A. and Schneider, K. (2011). Program sketching via CTL* model checking. In Groce, A. and Musuvathi, M., editors, *Model Checking Software (SPIN)*, volume 6823 of *LNCS*, pages 126–143, Snowbird, Utah, USA. Springer. v, 1, 4, 22, 28, 37, 39, 51

[20] Paulin-Mohring, C. and Werner, B. (1993). Synthesis of ml programs in the system coq. *Journal of Symbolic Computation*, 15(5):607 – 640. 2

[21] Pnueli, A. and Rosner, R. (1989). On the Synthesis of a Reactive Module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 179–190, New York, NY, USA. Association for Computing Machinery. 1, 3

[22] Sanghavi, A. (2010). What is formal verification? *EE Times-Asia*, pages 1–2. 7, 22

[23] Solar-Lezama, A. (2003). *Program Synthesis by Sketching.* PhD thesis, University of California, Berkeley, Berkeley, CA, US. 1, 2, 3, 21, 28, 36, 51

[24] Solar-Lezama, A. (2018). Introduction to Program Synthesis. `http://people.csail.mit.edu/asolar/SynthesisCourse/index.htm`. Accessed: 12-02-2020. ix, 1, 3, 21

[25] Vardi, M. Y. (2007). Automata-Theoretic Model Checking Revisited. In Cook, B. and Podelski, A., editors, *Verification, Model Checking, and Abstract Interpretation*, pages 137–150, Berlin, Heidelberg. Springer Berlin Heidelberg. 47